Lugaru's epsilon
programmer's editor

This is revision 12r of the manual.
It describes version 12.10 of Epsilon and EEL.

Lugaru Software Ltd.

Note to Our Users

For your convenience, we have not put any annoying copy protection mechanisms into Epsilon. We hope that you will respect our efforts, and the law, and not allow illegal copying of Epsilon.

Under the Copyright Law, if you provide a copy of Epsilon to anyone else for any reason, you lose the right to use it yourself. You may under no circumstances transfer the program or manual to more than one party. The Copyright Law says, in part:

> Any exact copies prepared in accordance with the provisions of this section may be leased, sold, or otherwise transferred, along with the copy from which such copies were prepared, only as part of the lease, sale, or other transfer of all rights in the program.

In other words, treat Epsilon like a book. If you sell your copy, don't keep a copy for yourself!

In addition, Lugaru grants the purchaser of Epsilon permission to install Epsilon on up to two computers at one time, as long as there is no possibility that Epsilon will be in use on more than one computer at a time. The end-user may, for example, install Epsilon on his or her computer at work and at home, as long as there is no possibility that Epsilon will be used on both computers at the same time. This permission applies to copies of Epsilon purchased by an end-user and not subject to a written license agreement.

We wish to thank all of our users who have made Epsilon successful, and extend our welcome to all new users.

Steven Doerfler
Lugaru Software, Ltd.

We produced this manual using the Epsilon Programmer's Editor and the TeX typesetting system. Duane Bibby did the illustrations.

Contents

Chapter 1

Welcome

1.1   Introduction

Welcome! We hope you enjoy using Epsilon. We think you'll find that Epsilon provides power and flexibility unmatched by any other editor for a personal computer.

Epsilon has a command set and general philosophy similar to the EMACS-style editors used on many different kinds of computers. If you've used an EMACS-style editor before, you will find Epsilon's most commonly used commands and keys familiar. If you haven't used an EMACS-style editor before, you can use Epsilon's tutorial program. Chapter 2 tells you how to install Epsilon and how to use the tutorial program.

1.2   Features

- Full screen editing with an EMACS-style command set.

- An exceptionally powerful embedded programming language, called EEL, that lets you customize or extend the editor. EEL provides most of the expressive power of the C programming language.

- The ability to invoke other programs from within Epsilon, in various ways. See page 118.

- The ability to run some classes of programs concurrently with the output going to a window. Details begin on page 119.

- You can invoke your compiler or "make" program from within Epsilon, then have Epsilon scan the output for error messages, then position you at the offending line in your source file. See page 121.

- An undo command that lets you "take back" your last command, or take back a sequence of commands. The undo facility works on both simple and complicated commands. Epsilon has a redo command as well, so you can even undo your undo's. See page 83.

- Very fast redisplay. We designed Epsilon specifically for the personal computer, so it takes advantage of the high available display bandwidth.

- Epsilon can dynamically syntax-highlight source code files written in many different languages, showing keywords in one color, functions in another, string constants in a third, and so forth.

- Epsilon can finish typing long identifier names for you.

- You can interactively rearrange the keyboard to suit your preferences, and save the layout so that Epsilon uses it the next time. Epsilon can also emulate the Brief text editor's commands, or use a CUA-style keyboard (like various Windows programs).

- You can edit a virtually unlimited number of files simultaneously.

- Epsilon understands Internet URL's and can asynchronously retrieve and send files via FTP. Telnet and related commands are also built in.

- Epsilon provides a multi-windowed editing environment, so you can view several files simultaneously. You can use as many windows as will fit on the display. See page 86.

- Under Windows, Epsilon provides a customizable tool bar.

- An extensive on-line help system. You can get help on what any command does, what any key does, and on what the command executing at the moment does. And Epsilon's help system will automatically know about any rearrangement you make to the keyboard. See page 31.

- An extensible "tags" system for many programming languages that remembers the locations of subroutine and variable definitions. You provide a subroutine name, for instance, and Epsilon takes you to the place that defines that subroutine. Alternatively, you can position the cursor on a function call, hit a key, and jump right to the definition of that function. See page 43.

- Completion on file names and command names. Epsilon will help you type the names of files and commands, and display lists of names that match a pattern that you specify. You can complete on many other classes of names too. This saves you a lot of typing. See page 24.

- Support for Unicode files and files using a variety of other character sets.

- Under Windows, you can drag and drop files or directories onto Epsilon's window, and Epsilon will open them.

- Commands to manipulate words, sentences, paragraphs, and parenthetic expressions. See the commands starting on page 36.

- Indenting and formatting commands. Details start on page 65.

- A kill ring to store text you've previously deleted. You can set the number of such items to save. See page 49.

- A convenient *incremental search* command (described on page 38), as well as regular searching commands, and search-and-replace commands.

- Regular expression searches. With regular expressions you can search for complex patterns, using such things as wildcards, character classes, alternation, and repeating.

- A fast *grep* command that lets you search across a set of files. See page 41. You can also replace text in a set of files.

- Extended file patterns that let you easily search out files on a disk.

- A directory editing command that lets you navigate among directories, copying, moving, and deleting files as needed. It even works on remote directories via FTP.

- Fast *sort* commands that let you quickly sort a buffer. See page 64.

- A powerful *keyboard macro* facility (see page 124), that allows you to execute sequences of keystrokes as a unit, and to extend the command set of the editor. You'll find Epsilon's keyboard macros very easy to define and use.

- Commands to compare two files and find the differences between them. You can compare character-by-character or line-by-line. See page 45.

- You can choose from a variety of built-in screen layouts, making Epsilon's screen look like those of other editors, or customize your own look for the editor.

Chapter 2

Getting Started

This chapter tells you how to install Epsilon on your system and explains how to invoke Epsilon. We also describe how to run the tutorial, and list the files in an Epsilon distribution.

2.1   Installing Epsilon for Windows

Epsilon for Windows is provided as a self-installing Windows executable. Run the program

```
r:\setup.exe
```

where r represents your CD-ROM drive. You can also use Add/Remove Programs in the Control Panel.

The installation program installs the GUI version of Epsilon for Windows, and the Win32 console version. We named the Windows GUI version `epsilon.exe` and the console version `epsilonc.exe`.

The installation program creates program items to run Epsilon. It also sets the registry entry Software\Lugaru\Epsilon\EpsPath*version* in the `HKEY_CURRENT_USER` hierarchy to include the name of the directory in which you installed Epsilon (where *version* represents Epsilon's version number).

Under Windows 95/98/ME, the installation program directs the system to install Epsilon's VxD each time it starts, by creating the registry entry System\CurrentControlSet\Services\VxD\Epsilon*version*\StaticVxD in the `HKEY_LOCAL_MACHINE` hierarchy. If you're running Windows 95/98/ME, the program will warn that you must restart Windows before the concurrent process will work.

You can uninstall Epsilon by using Add/Remove Programs in the Control Panel.

2.2   Installing Epsilon for Unix

Epsilon includes a version for Linux and a separate version for FreeBSD. We describe them collectively as the "Unix" version of Epsilon. To install either one, mount the CD-ROM, typically by typing

```
mount /cdrom
```

Then, as root, run the appropriate shell script. For Linux, that would be

```
/cdrom/linux/einstall
```

and for FreeBSD that would be

```
/cdrom/freebsd/einstall
```

The installation script will prompt you for any necessary information.

(Under Linux, you may need to provide the `-o exec` option to the `mount` command.)

If for some reason that doesn't work, you can manually perform the few steps needed to install Epsilon. For Epsilon for Linux, you would type, as root:

```
cd /usr/local
tar xzf /cdrom/linux/epsilon12.10.tar.gz
cd epsilon12.10
./esetup
```

For FreeBSD, substitute `freebsd` for `linux` in the second command.

You can also install Epsilon in a private directory, if you don't have root access. In that case you will also need to define an environment variable so Epsilon can locate its files, such as

```
EPSPATH1210=~/.epsilon:/home/bob/epsilon12.10
```

If you install Epsilon in a private directory, the `esetup` command will display the environment variable definition you'll need.

Some versions of Epsilon use a helper program to access certain shared library files from the glibc 2.1 NSS subsystem. If necessary, the installation script will compile a helper program to provide Epsilon with these services.

Epsilon runs as an X program with X and as a text program outside of X. Epsilon knows to use X when it inherits a `DISPLAY` environment variable. You can override Epsilon's determination by providing a `-vt` flag to make Epsilon run as a text program, or an appropriate `-display` flag to make Epsilon connect to a given X server.

Epsilon also recognizes these standard X flags:

-bw *pixels* **or -borderwidth** *pixels* This flag sets the width of the window border in pixels. An `Epsilon.borderWidth` resource may be used instead.

-fn *font* **or -font** *font* This flag specifies the font to use. The Alt-x set-font command can select a different font from within Epsilon. Epsilon will remember any font you select with set-font and use it in future sessions; this flag overrides any remembered font.

-geometry *geometry* This flag sets the window size and position, using the standard X syntax. Without this flag, Epsilon looks for an `Epsilon.geometry` resource.

-name *resname* This flag tells Epsilon to look for X resources using a name other than "Epsilon".

-title *title* This flag sets the title Epsilon displays while starting. An `Epsilon.title` resource may be used instead.

-xrm *resourcestring* This flag specifies a specific resource name and value, overriding any defaults.

Epsilon uses various X resources. You can set them from the command line with a flag like `-xrm Epsilon.cursorstyle:1` or put a line like `Epsilon.cursorstyle:1` in your X resources file, which is usually named `~/.Xresources` or `~/.Xdefaults`:

```
Epsilon.cursorstyle: 1
```

You'll need to tell X to reread the file after making such a change, using a command like `xrdb -merge ~/.Xresources`.

Epsilon uses these X resources:

Epsilon.borderWidth This sets the width of the border around Epsilon's window.

Epsilon.cursorstyle Under X, Epsilon displays a block cursor whose shape does not change. Define a cursorstyle resource with value 1 and Epsilon will use a line-style cursor, sized to reflect overwrite mode or virtual space mode. Note this cursor style does not display correctly on some older X servers.

Epsilon.font This resource sets Epsilon's font. It must be a fixed-width font. If you set a font from within Epsilon, it remembers your selection in a file `~/.epsilon/Xresources` and uses it in future sessions. Epsilon uses this resource if there's no font setting in that file.

Epsilon.geometry This resource provides a geometry setting for Epsilon. See the `-geometry` flag above.

Epsilon.title This resource sets the title Epsilon displays while starting.

2.3 Installing Epsilon for DOS

An older version of Epsilon for DOS is also provided on the CD-ROM, for users who must use DOS.

The Win32 console version, described previously, and the DOS version have a similar appearance, and both will run in Windows, but of the two, only the Win32 console version can use long file names or the clipboard in all 32-bit versions of Windows. The DOS version also lacks a number of other features in the Win32 console version. If you wish to run Epsilon from a command line prompt (a DOS box) within any 32-bit version of Windows, use the Win32 console version, not the DOS version, for the best performance and feature set.

To install Epsilon for DOS, cd to the `\DOS` directory on the Epsilon CD-ROM. Run Epsilon's installation program by typing:

```
install
```

Follow the directions on the screen to install Epsilon. The installation program will ask before it modifies or replaces any system files. The DOS executable is named epsdos.exe. A list of files provided with Epsilon starts on page 15.

2.4 Installing Epsilon for OS/2

An older version of Epsilon for OS/2 is also provided on the CD-ROM. To install Epsilon for OS/2, start a command prompt and cd to the `\OS2` directory on the Epsilon CD-ROM. Run Epsilon's installation program by typing:

```
install
```

Follow the directions on the screen to install Epsilon. The installation program will ask before it modifies or replaces any system files. The OS/2 executable is named epsilon.exe. A list of files provided with Epsilon starts on page 15.

2.5 Tutorial

Once you install Epsilon, put the distribution medium away. If you've never used Epsilon or EMACS before, you should run the tutorial to become acquainted with some of Epsilon's simpler commands.

The easiest way to run the tutorial is to start Epsilon and select Epsilon Tutorial from the Help menu. (If you're running a version of Epsilon without a menu bar, you can instead press the F2 key in Epsilon and type the command name `tutorial`. Or you can start Epsilon with the -teach flag.)

The tutorial will tell you everything else you need to know to use the tutorial, including how to exit the tutorial.

2.6 Invoking Epsilon

You can start Epsilon for Windows using the icon created by the installer. Under other operating systems, you can run Epsilon by simply typing "epsilon".

Depending on your installation options, you can also run Epsilon for Windows from the command line. Under Windows, type "epsilon" to run the more graphical version of Epsilon, or "epsilonc" to run the Win32 console version of Epsilon. "Epsdos" runs the DOS version, if one is installed.

The first time you run Epsilon, you will get a single window containing an empty document. You can give Epsilon the name of a file to edit on the command line. For example, if you type

```
epsilon sample.c
```

then Epsilon will start up and read in the file `sample.c`. If the file name contains spaces, surround the entire name with double-quote characters.

```
epsilon "a sample file.c"
```

When you name several files on the command line, Epsilon reads each one in, but puts only up to three in windows (so as not to clutter the screen with tiny windows). You can set this number by modifying the `max-initial-windows` variable.

If you specify files on the command line with wild cards, Epsilon will show you a list of the files that match the pattern in dired mode. See page 110 for more information on how dired works. File names that contain only extended wildcard characters like , ; [or], and no standard wildcard characters like * or ?, will be interpreted as file names, not file patterns. (If you set the variable `expand-wildcards` to 1, Epsilon will instead read in each file that matches the pattern, as if you had listed them explicitly. Epsilon for Unix does this too unless you quote the file pattern.)

Epsilon normally shows you the beginning of each file you name on the command line. If you want to start at a different line, put "+*number*" before the file's name, where *number* indicates the line number to go to. You can follow the line number with a :column number too. For example, if you typed

```
epsilon +26 file.one +144:20 file.two
```

then you would get file.one with the cursor at the start of line 26, and file.two with the cursor at line 144, column 20.

By default, Epsilon will also read any files you were editing in your previous editing session, in addition to those you name on the command line. See page 113 for details.

If you're running an evaluation version of Epsilon or a beta test version, you may receive a warning message at startup indicating that soon your copy of Epsilon will expire. You can disable or delay this warning message (though not the expiration itself). Create a file named `no-expiration-warning` in Epsilon's main directory. Put in it the maximum number of days warning you want before expiration.

2.7   Configuration Variables: The Environment and The Registry

Epsilon for Unix uses several environment variables to set options and say where to look for files. Epsilon for Windows stores such settings in the System Registry, under the key `HKEY_CURRENT_USER\SOFTWARE\Lugaru\Epsilon`. Epsilon's setup program will generally create all necessary registry keys automatically.

We use the term *configuration variable* to refer to any setting that appears as an environment variable under Unix, or a registry entry under Windows. There are a small number of settings that are stored in environment variables on all platforms; these are generally settings that are provided by the operating system. These include COMSPEC, TMP or TEMP, EPSRUNS, and MIXEDCASEDRIVES.

Under Windows, the installation program creates a registry entry similar to this:

```
HKEY_CURRENT_USER\SOFTWARE\Lugaru\Epsilon\EpsPath=~;c:\epsilon
```

Of course, the actual entry, whether it's an environment variable setting or an entry in the system registry, would contain whatever directory Epsilon was actually installed in, not c:\epsilon.

If you have more than one version of Epsilon on your computer, you may want each to use a different set of options. You can override many of the configuration variables listed below by using a configuration variable whose name includes the specific version of Epsilon in use. For example, when Epsilon needs to locate its help file, it normally uses a configuration variable named EPSPATH. Epsilon version 6.01 would first check to see if a configuration variable named EPSPATH601 existed. If so, it would use that variable. If not, it would then try EPSPATH60, then EPSPATH6, and finally EPSPATH. Epsilon does the same sort of thing with all the configuration variables it uses, with the exception of DISPLAY, EPSRUNS, TEMP, and TMP.

Epsilon uses a similar procedure to distinguish registry entries for the Win32 console mode version from registry entries for the Win32 GUI version of Epsilon. For the console version, it checks registry names with an -NTCON suffix before the actual names; for the GUI version it checks for a -WIN suffix. So Epsilon 10.2 for Win32 console would seek an EPSPATH configuration variable using the names EPSPATH102-NTCON, EPSPATH102, EPSPATH10-NTCON, EPSPATH10, EPSPATH-NTCON, and finally EPSPATH, using the first one it finds.

For example, the Windows installation program for Epsilon doesn't actually add the EPSPATH entry shown above to the system registry. It really uses an entry like

```
HKEY_CURRENT_USER\SOFTWARE\Lugaru\Epsilon\EpsPath80=c:\epsilon
```

where EpsPath80 indicates that the entry should be used by version 8.0 of Epsilon, or version 8.01, or 8.02, but not by version 8.5. In this way, multiple versions of Epsilon can be installed at once, without overwriting each other's settings. This can be helpful when upgrading Epsilon from one version to the next.

Here we list all the configuration variables that Epsilon can use. Remember, under Windows, most of these names refer to entries in the registry, as described above. Under Unix, these are all environment variables.

CMDCONCURSHELLFLAGS If defined, Epsilon puts the contents of this variable before the command line when you use the start-process command with a numeric argument. It overrides CMDSHELLFLAGS. See page 119.

CMDSHELLFLAGS If defined, Epsilon puts the contents of this variable before the command line when it runs a subshell that should execute a single command and exit.

COMSPEC Epsilon for Windows needs a valid COMSPEC environment variable in order to run another program. Normally, the operating system automatically sets up this variable to give the file name of your command processor. If you change the variable manually, remember that the file must actually exist. Don't include command line options for your command processor in the COMSPEC variable. If a configuration variable called EPSCOMSPEC exists, Epsilon will use that instead of COMSPEC. (For Unix, see SHELL below.)

DISPLAY Epsilon for Unix tries to run as an X program if this environment variable is defined, using the X server display it specifies.

EEL The EEL compiler looks for a configuration variable named EEL before examining its command line, then "types in" the contents of that variable before the compiler's real command line. See page 155.

EPSCOMSPEC See COMSPEC above.

EPSCONCURCOMSPEC If defined, Epsilon for Windows runs the shell command processor named by this variable instead of the one named by the EPSCOMSPEC or COMSPEC variables, when it starts a concurrent process. See page 119.

EPSCONCURSHELL If defined, Epsilon for Unix runs the shell command processor named by this variable instead of the one named by the EPSSHELL or SHELL variables, when it starts a concurrent process. See page 119.

EPSILON Before examining the command line, Epsilon looks for a configuration variable named EPSILON and "types in" the value of that variable to the command line before the real command line. See page 11.

EPSMIXEDCASEDRIVES This variable can contain a list of drive letters. If the variable exists, Epsilon doesn't change the case of file names on the listed drives. See page 102 for details.

EPSPATH Epsilon uses this configuration variable to locate its files. See page 10.

EPSRUNS When Epsilon runs another program, it sets this environment variable to indicate to the other program that it's running within Epsilon. A setting of C indicates the subprocess is running within Epsilon's concurrent process. A setting of P indicates the subprocess is running via the filter-region command or similar. A setting of Y indicates Epsilon ran the process in some other way, such as via the shell command.

EPSSHELL See SHELL below.

ESESSION Epsilon uses this variable as the name of its session file. See page 113.

INTERCONCURSHELLFLAGS If defined, Epsilon uses the contents of this variable as the command line to the shell command processor it starts when you use the start-process command without a numeric argument. It overrides INTERSHELLFLAGS. See page 119.

INTERSHELLFLAGS If defined, Epsilon uses the contents of this variable as a subshell command line when it runs a subshell that should prompt for a series of commands to execute. See page 119.

MIXEDCASEDRIVES This variable can contain a list of drive letters. If the variable exists, Epsilon doesn't change the case of file names on the listed drives. See page 102 for details.

PATH The operating system uses this variable to find executable programs such as epsilon.exe. Make sure this variable includes the directory containing Epsilon's executable files if you want to conveniently run Epsilon from the command line.

SHELL Epsilon for Unix needs a valid SHELL environment variable in order to run another program. If a configuration variable called EPSSHELL exists, Epsilon will use that instead of SHELL. (See COMSPEC above for the non-Unix equivalent.)

TEMP Epsilon puts any temporary files it creates in this directory, unless a TMP environment variable exists. See the description of the −fs flag on page 12.

TMP Epsilon puts any temporary files it creates in this directory. See the description of the −fs flag on page 12.

2.7.1   How Epsilon Finds its Files

Sometimes Epsilon needs to locate one of its files. For example, Epsilon needs to read an .mnu file like gui.mnu or epsilon.mnu to determine what commands go in its menu bar. Epsilon searches directories in this order:

- The directory containing the Epsilon executable. The −w4 flag tells Epsilon to skip this step.

- The parent of the directory containing the Epsilon executable. The -w8 flag tells Epsilon to skip this step. Epsilon also skips this step under Unix, or if the name of the directory with Epsilon's executable doesn't start with `bin`, or its parent doesn't start with `eps`. They do in a default installation.

- The directories specified by the EPSPATH configuration variable. This configuration variable should contain a list of directories separated by semicolons (or for Unix, colons). Epsilon will then look for the file in each of these directories.

Under Windows, a directory named ~ in an EPSPATH variable has a special meaning. It refers to the current user's customization directory. See the next section.

If there is no EPSPATH configuration variable, Epsilon constructs a default one. For Windows, the default EPSPATH is just ~, the user's configuration directory. For Unix, the default EPSPATH contains ~/.epsilon, then /usr/local/epsilon*VER* (where *VER* indicates the current version, such as 10.01), then /usr/local/epsilon and then /opt/epsilon.

The EEL compiler also uses the EPSPATH environment variable. See page 155.

2.7.2   The Customization Directory

Epsilon searches for some files in a user-specific customization directory. It also creates files like its initialization file `einit.ecm` there.

Under Linux, the customization directory is always `~/.epsilon`.

Under Windows, the customization directory is located in the `Lugaru\Epsilon` subdirectory within the current user's Application Data directory, which varies by version of Windows. Here are some typical locations:

For Windows 2000/XP:

`\Documents and Settings\`*username*`\Application Data\Lugaru\Epsilon`

For Windows NT:

`\Winnt\Profiles\`*username*`\Application Data\Lugaru\Epsilon`

For Windows 95/98/ME, when user login is enabled:

`\Windows\Profiles\`*username*`\Application Data\Lugaru\Epsilon`

For Windows 95/98/ME, when user login is disabled:

`\Windows\Application Data\Lugaru\Epsilon`

You can use the edit-customizations command to locate your customization directory.

2.8   Epsilon Command Line Flags

When you start Epsilon, you may specify a sequence of command line flags (also known as command-line options, or switches) to alter Epsilon's behavior. Flags must go before any file name.

Each flag consists of a minus sign ("-"), a letter, and sometimes a parameter. You can use the special flag -- to mark the end of the flags; anything that follows will be interpreted as a file name even if it starts with a - like a flag.

If a parameter is required, you can include a space before it or not. If a parameter is optional (-b, -m, -p) it must immediately follow the flag, with no space.

Before examining the command line, Epsilon looks for a configuration variable (see page 8) named
EPSILON and "types in" the value of that variable to the command line before the real command line. Thus,
if you define a Unix environment variable:

```
export EPSILON=-m250000 -smine
```

then Epsilon would behave as if you had typed

```
epsilon -m250000 -smine myfile
```

when you actually type

```
epsilon myfile
```

Here we list all of the flags, and what they do:

+*number* Epsilon normally shows you the beginning of each file you name on the command line. If you
    want to start at a different line, put "**+***number*" before the file's name, where *number* indicates the line
    number to go to. You can follow the line number with a colon and a column number if you wish.

−add This flag tells Epsilon to locate an existing instance of Epsilon, pass it the rest of the command line,
    and exit. Epsilon ignores the flag if there's no prior instance. If you want to configure another
    program to run Epsilon to edit a file, but use an existing instance of Epsilon if there is one, just include
    this flag in the Epsilon command line. See page 116 for details on Epsilon's server support.

−b*filename* Epsilon normally reads all its commands from a state file at startup. (See the −s flag below.)
    Alternately, you can have Epsilon start up from a file generated directly by the EEL compiler. These
    bytecode files end with a ".b" extension. This flag says to use the bytecode file with name *filename*, or
    "epsilon" if you leave out the *filename*. You may omit the extension in *filename*. You would rarely use
    this flag, except when building a new version of Epsilon from scratch. Compare the −l flag.

−d*variable!value* You can use this flag to set the values of string and integer variables from the command
    line. The indicated variable must already exist at startup. You can also use the syntax
    −d*variable=value*, but beware: if you run Epsilon for Windows via a .BAT or .CMD file, the system
    will replace any ='s with spaces, and Epsilon will not correctly interpret the flag.

−fd*filename* This flag tells Epsilon where to look for the on-line documentation file. Normally, Epsilon
    looks for a file named edoc. This flag tells Epsilon to use *filename* for the documentation file. If you
    provide a relative name for *filename*, then Epsilon will search for it; see page 10. Use a file name, not
    a directory name, for *filename*.

−fs*dirnames* This switch tells Epsilon what directories to use for temporary files, such as Epsilon's swap
    file, which it uses when you edit files too big for available memory, or the eshell file it creates in some
    environments to help capture the output of a process. *Dirnames* should indicate a list of one or more
    directories, separated by semicolons (colons under Unix). Epsilon will use the first directory named as
    long as there is space on its device; then it will switch to the second directory, and so forth. If it cannot
    find any available space, it will ask you for another directory name.

    If you don't use this switch, Epsilon will create any temporary files it needs in the directory named by
    the TMP environment variable. If TMP doesn't exist, Epsilon tries TEMP, then picks a fallback
    location. Epsilon calls its swap file eswap, but it will use another name (like eswap0, eswap1, etc.) to
    avoid a conflict with another Epsilon using this file.

−geometry When Epsilon for Unix runs as an X program, it recognizes this standard X flag.

-ka*number* (Windows only) This switch turns off certain keyboard functions to help diagnose problems. It's followed by a number. -ka1 tells Epsilon not to translate the Ctrl-2 key combination to NUL. -ka4 tells Epsilon to try to work around a rare caret-related screen painting bug on some display cards. -ka8 tells Epsilon to be more conservative when writing text on the screen, at the price of some performance; it may help with fonts that use inconsistent character sizes. -ka16 is similar, but has less effect on performance.

-ks*number* This flag lets you adjust the emphasis Epsilon puts on speed during long operations versus responsiveness to the abort key. Higher numbers make Epsilon slightly faster overall, but when you press the abort key, Epsilon may not respond as quickly. Lower numbers make Epsilon respond more quickly to the abort key, but with a performance penalty. The default setting is -ks100.

-l*bytecode* Giving this switch makes Epsilon load a bytecode file named *bytecode*.b after loading the state file. If you give more than one -l flag on the command line, the files load in the order they appear. Compare the -b flag.

-m*bytes* This switch controls how much memory Epsilon uses for the text of buffers. Epsilon interprets a number less than 1000 as a number of kilobytes, otherwise, as bytes. You may explicitly specify kilobytes by ending *bytes* with 'k', or megabytes by ending *bytes* with 'm'. Specify -m0 to use as little memory as possible, and -m to put no limit on memory use.

If you read in more files than will fit in the specified amount of memory, or if despite a high limit, the operating system refuses Epsilon's requests for more memory, Epsilon will swap portions of the files to disk. By default, Epsilon puts no limits on its own memory usage.

-noinit This flag tells Epsilon not to read any `einit.ecm` customization file.

-nologo In some environments Epsilon prints a short copyright message when it starts. This flag makes it skip displaying that message.

-noserver This flag tells Epsilon for Windows or Unix that it should not register itself as a server so as to accept messages from other instances of Epsilon. By default, Epsilon will receive messages from future instances of Epsilon that are started with the -add flag, or (for Windows) sent via file associations or DDE. See page 116 for details. The flag -nodde is a synonym.

-p*filename* This overrides the ESESSION configuration variable to control the name of the session file that Epsilon uses. When you specify a file name, Epsilon uses that for the session file, just as with ESESSION. Because the -p0 and -p1 flags enable and disable sessions (see the next item), the given *filename* must not begin with a digit.

-p*number* This flag controls whether or not Epsilon restores your previous session when it starts up. By default, Epsilon will try to restore your previous window and buffer configuration. The -p flag with no number toggles whether Epsilon restores the session. Give the -p0 flag to disable session restoring and saving, and the -p1 flag to enable session restoring and saving. This flag understands the same values as the `preserve-session` variable; see its description for other options.

-quickup Epsilon uses this flag to help perform certain updates. It searches for and loads a bytecode file named quickup.b. This flag is similar to the -l flag above, but the -quickup flag doesn't require any EEL functions to run. For that reason, it can replace and update any EEL function.

-r*command* Giving this switch makes Epsilon try to run a command or keyboard macro named *command* at startup. If the command doesn't exist, nothing happens. If you specify more than one -r flag on the command line, they execute in the order they appear. Use the syntax -r*cmdname*=*param* or -r*cmdname*!*param* to run an EEL subroutine and pass it a value; the subroutine must be defined to accept a single parameter of `char *` type.

−s*filename* When Epsilon starts up, it looks for a *state file* named epsilon.sta. The state file contains
definitions for all of Epsilon's commands. You can create your own state file by using the write-state
command. This switch says to use the state file with the name *filename*. Epsilon will add the
appropriate extension if you omit it. Specify a file name for *filename*, **not** a directory name. Of course,
the file name may include a directory or drive prefix. If you specify a relative file name, Epsilon will
search for it. See page 10. See also the −b flag, described above.

−server:*servername* The command line flag −server may be used to alter the server name for an instance of
Epsilon. An instance of Epsilon started with −server:*somename* −add will only pass its command line
to a previous instance started with the same −server:*somename* flag. See page 116. The flag −dde is a
synonym.

−teach This flag tells Epsilon to load the on-line tutorial file at startup. See page 7.

−vc*x* *x* indicates the number of columns you want displayed while in Epsilon. For example, use "−vc132"
for 132 columns. See the −vl flag, described below. See the −geometry flag for the equivalent in
Epsilon for Unix.

−vcolor Epsilon normally tries to determine whether to use a monochrome color scheme or a full-color one
based on the type of display in use and its mode. This flag forces Epsilon to use a full-color color
scheme, regardless of the type of the display.

−vl*x* *x* indicates the number of screen lines you want to use while in Epsilon. Also See the −vc switch,
described above. See −geometry for the equivalent in Epsilon for Unix.

−vmono Epsilon normally tries to determine whether to use a monochrome color scheme or a full-color one
based on the type of display in use and its mode. This flag forces Epsilon to use its monochrome color
scheme, regardless of the type of the display.

−vt (Unix only) This flag forces Epsilon to run as a curses-style terminal program, not an X program. By
default Epsilon for Unix runs as an X program whenever an X display is specified (either through a
DISPLAY environment variable or a −display flag), and a terminal program otherwise.

−vv This flag instructs Epsilon to split the screen vertically, not horizontally, when more than one file is
specified on the command line.

−vx and −vy These flags let you specify the position of Epsilon's window in Epsilon for Windows. For
example, −vx20 −vy30 positions the upper left corner of Epsilon's window at pixel coordinates
20x30. See −geometry for the equivalent in Epsilon for Unix.

−w*number* This flag controls several directory-related settings. Follow it with a number.

The −w1 flag tells Epsilon to remember the current directory from session to session. Without this
flag, Epsilon will remain in whatever current directory it was started from. Epsilon always records the
current directory when it writes a session file; this flag only affects whether or not Epsilon uses this
information when reading a session file.

The −w2 flag has no effect in this version of Epsilon.

The −w4 flag tells Epsilon not to look for its own files in the directory containing the Epsilon
executable. Similarly, the −w8 flag tells Epsilon not to look for its own files in the parent of the
directory containing the Epsilon executable. Epsilon normally looks for its own files in these two
directories, prior to searching the EPSPATH. If you choose to put Epsilon's executable files in a
common directory with other executable files, you may wish to set this flag. If you do this, make sure
the EPSPATH points to the correct directory.

The −w16 flag tells Epsilon to set its current directory to the directory containing the first file named on its command line. If you edit files by dragging and dropping them onto a shortcut to Epsilon, you may wish to use this flag in the shortcut.

You can combine −w flags by adding their values together. For example, −w5 makes Epsilon remember the current directory and exclude its executable's directory from the EPSPATH. All Windows program icons for Epsilon invoke it with −w1 so that Epsilon remembers the current directory. (When you tell the installer to put Epsilon's executables in a common directory, not in Epsilon's normal directory structure, the installer uses the −w13 flag in Epsilon's icons, and the −w12 flag when Epsilon is invoked from the command line. Epsilon then relies on the EPSPATH setting to find its files.)

−**wait** This flag tells Epsilon for Unix to locate an existing instance of Epsilon, pass it the rest of the command line, and wait for the user in that instance to invoke the resume-client command. (Epsilon ignores the flag if there's no prior instance.) If you want to configure another program to run Epsilon to edit a file, but use an existing instance of Epsilon, just include this flag in the Epsilon command line. See page 116 for details on Epsilon's server support.

2.9 File Inventory

Epsilon consists of the following files:

setup.exe, setup.w02 (Windows only) Epsilon's installation program.

epsilon.exe The 32-bit Epsilon for Windows executable program.

epsilonc.exe The Epsilon executable program for Win32 console mode.

epsdos.exe The Epsilon executable program for DOS-only systems.

epsdos.ico and epsdos.pif These files help the DOS version of Epsilon to run under Windows.

eel.exe Epsilon's compiler. You need this program if you wish to add new commands to Epsilon or modify existing ones.

eel_lib.dll Under Windows, Epsilon's compiler eel.exe requires this file. Epsilon itself also uses this file when you compile from within the editor.

icudt*.dat, eunicode.dll These files help provide Unicode support.

conagent.pif, concur16.exe, concur16.ico, and concur16.pif Epsilon for Windows requires these files to provide its concurrent process feature.

lugeps1.386 Epsilon for Windows requires this file under Windows 95/98/ME to provide its concurrent process feature. It's normally installed in your Windows System directory.

inherit.exe and inherit.pif Epsilon for Windows uses these files to execute another program and capture its output.

sheller.exe and sheller.pif Epsilon for Windows 95/98/ME uses these files as well to execute another program and capture its output.

edoc.hlp This Windows help file provides help on Epsilon.

epshlp.dll Epsilon's help file communicates with a running copy of Epsilon so it can display current key bindings or variable values and let you modify variables from the help file. It uses this file to do that.

sendeps.exe Epsilon for Windows uses this file to help create desktop shortcuts to Epsilon, or Send To
menu entries.

VisEpsil.dll Epsilon for Windows includes this Developer Studio extension that lets Developer Studio pass
all file-opening requests to Epsilon.

The installation program puts the following files in the main Epsilon directory, normally \Program
Files\Eps12under Windows and /usr/local/epsilon12.10 under Unix.

epsilon.sta This file contains all of Epsilon's commands. Epsilon needs this file in order to run. If you
customize Epsilon, this file changes. Epsilon for Unix includes a version number in this name.

original.sta This file contains a copy of the original version of epsilon.sta at the time of installation.

edoc Epsilon's on-line documentation file. Without this file, Epsilon can't provide basic help on commands
and variables.

info\epsilon.inf Epsilon's on-line manual, in Info format.

info\dir A default top-level Info directory, for non-Unix systems that may lack one. See Info mode for
details.

lhelp* This directory contains files for the HTML version of Epsilon's documentation. The lhelp helper
program reads them.

epswhlp.hlp and epswhlp.cnt Epsilon uses these files to provide its search-all-help-files command under
Windows.

eteach Epsilon's tutorial. Epsilon needs this file to give the tutorial (see page 7). Otherwise, Epsilon does
not need this file to run.

colclass.txt One-line descriptions of each of the different color classes in Epsilon. The set-color command
reads this file.

brief.kbd The brief-keyboard command loads this file. It contains the bindings of all the keys used in Brief
emulation, written in Epsilon's command file format.

epsilon.kbd The epsilon-keyboard command loads this file. It contains the standard Epsilon key bindings
for all the keys that are different under Brief emulation, written in Epsilon's command file format.

epsilon.mnu Epsilon for Unix uses this file to construct its menu bar, except in Brief mode.

brief.mnu In Brief mode, Epsilon for Unix uses this file to construct its menu bar.

gui.mnu Epsilon for Windows uses this file to construct its menu bar.

latex.env The tex-environment command in LaTeX mode (Alt-Shift-E) gets its list of environments from
this file. You can add new environments by editing this file.

lugaru.url This file contains a link to Lugaru's World Wide Web site. If you have an Internet browser
installed under Windows, you can open this file via its file association and connect to Lugaru's Web
site. The view-lugaru-web-site command uses this file.

primlist.doc This file alphabetically lists all the EEL primitive functions and variables.

readme.txt This file contains miscellaneous notes, and describes any features or files we added after we
printed this manual. You can use the Alt-x release-notes command to read it.

unwise.exe, unwise.ini If you used the Windows-based installer, you can uninstall Epsilon by running this program.

install.log The Windows-based installer creates this file to indicate which files it installed. Uninstalling Epsilon requires this file.

*.h The installation program copies a number of "include files" to the subdirectory "include" within Epsilon's main directory. These header files are used if you decide to compile an Epsilon extension or add-on written in its EEL extension language.

eel.h Epsilon's standard header file, for use with the EEL compiler.

codes.h Another standard header file, with numeric codes. The eel.h file includes this one automatically.

filter.h A header file defining the contents of Epsilon's Common File Open/Save dialogs under Windows.

*.e These files contain source code in EEL to all Epsilon's commands. The installation program copies them to the subdirectory "source" within Epsilon's main directory.

epsilon.e This file loads all the other files and sets up Epsilon.

makefile You can use this file, along with a "make" utility program, to help recompile the above Epsilon source files. It lists the source files and provides command lines to compile them.

The directory "changes" within Epsilon's main directory contains files that list the differences between this version of Epsilon and previous versions.

Chapter 3

General Concepts

This chapter describes the framework within which the commands operate. The chapter entitled "Commands by Topic", which starts on page 31, goes into detail about every Epsilon command.

If you have never used Epsilon before, you should run the tutorial now. This chapter discusses some general facilities and concepts used throughout Epsilon by many of the commands. You will find the discussion much clearer if you've used the tutorial, and have become accustomed to Epsilon's general style.

To run the tutorial, start Epsilon and select Epsilon Tutorial from the Help menu. (You can also press the F2 key in Epsilon and type the command name `tutorial`, or start Epsilon with the `-teach` flag.)

3.1   Buffers

In Epsilon's terminology, a *buffer* contains text that you can edit. You can think of a buffer as Epsilon's copy of a file that you have open for editing. Actually, a buffer may contain a copy of a file, or it may contain a new "file" that you've created but have not yet saved to disk.

To edit a file, you read the file into a buffer, modify the text of the buffer, and write the buffer to the file. A buffer need not necessarily correspond to a file, however. Imagine you want to write a short program from scratch. You fire up Epsilon, type the text of the program into a buffer, then save the buffer to a file.

Epsilon does not place any limitation on the number of active buffers during an editing session. You can edit as many buffers at the same time as you want. This implies that you can edit as many files, or create as many files, or both, as you desire. Each document or program or file appears in its own buffer.

3.2   Windows

Epsilon displays your buffers to you in *windows*. You can have one window or many windows. You can change the number and size of windows at any time. You may size a window to occupy the entire display, or to occupy as little space as one character wide by one character high.

Each window can display any buffer. You decide what a window displays. You can always get rid of a window without worrying about losing the information the window displays: deleting a window does **not** delete the buffer it displays.

Each window displays some buffer, and several windows can each display the same buffer. This comes in handy if you want to look at different parts of a buffer at the same time, say the beginning and end of a large file.

A buffer exists whether or not it appears in some window. Suppose a window displays a buffer, and you decide to refer to another file. You can read that file into the current window without disturbing the old buffer. You peruse the new buffer, then return to the old buffer.

You may find this scheme quite convenient. You have flexibility to arrange your buffers however you like on the screen. You can make many windows on the screen to show any of your buffer(s), and delete windows as appropriate to facilitate your editing. You never have to worry about losing your buffers by deleting or changing your windows.

Epsilon has many commands to deal with buffers and windows, such as creating, deleting, and changing the size of windows, reading files into a buffer, writing buffers out to files, creating and deleting buffers, and much more. We describe these in detail in the chapter "Commands by Topic", which starts on page 31.

3.3   Epsilon's Screen Layout

To see what buffers and windows look like, refer to figure 3.1. This shows what the screen looks like with only one window. It shows what the screen looks like when you edit a file named screen.1.
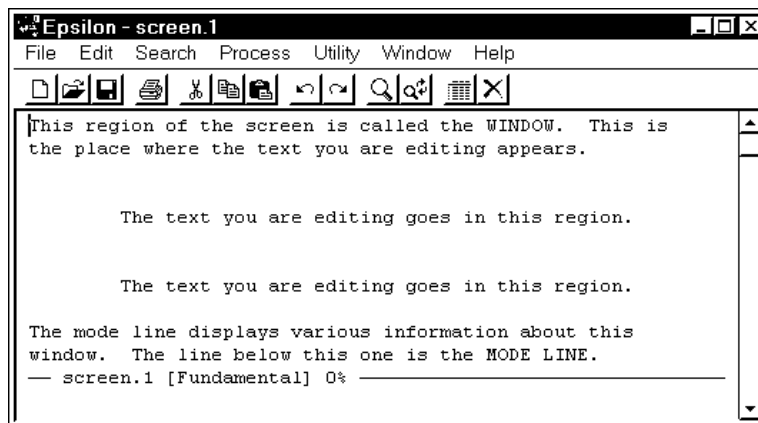
Figure 3.1: What Epsilon looks like with one window.

The top section of the screen displays some of the text of the window's buffer. Below that appears the *mode line*. The mode line shows the buffer name in parentheses, the file associated with that buffer, if any, the current *mode* in square brackets (see below), and the percentage of the buffer before the cursor. If the file and buffer have the same name (often the case), then the name appears only once in the mode line. A star (*) at the end of the line means that you have changed the buffer since the last time you saved it to disk. (See the `mode-format` variable for information on customizing the contents of the mode line.) The text area and the mode line collectively constitute the window.

Below the mode line, on the last line of the screen, appears the *echo area*. Epsilon uses this area to prompt you for information or to display messages (in the figure it's empty). For example, the command to read a file into a buffer uses the echo area to ask you for the file name. Regardless of how many windows you have on the screen, the echo area always occupies the bottommost screen line.

When Epsilon displays a message in the echo area, it also records the message in the `#messages#` buffer (except for certain transient messages). See the `message-history-size` variable to set how Epsilon keeps the buffer from excessive size by dropping old messages.

Epsilon has an important concept called the editing point, or simply *point*. While editing a buffer, the editing point refers to the place that editing "happens", as indicated by the cursor. Point refers not to a character position, but rather to a character *boundary*, a place *between* characters. You can think of point as, roughly, the leftmost edge of the cursor. Defining the editing point as a position between characters rather than at a particular character avoids certain ambiguities inherent in the latter definition.

Consider, for example, the command that goes to the end of a word, forward-word. Since point always refers to a position between characters, point moves right after the last letter in the word. So the cursor itself would appear underneath the first character after the word. The command that moves to the beginning of the word, backward-word, positions point right before the first character in the word. In this case, the cursor itself would appear under the first character in the word.

When you want to specify a region, this definition for point avoids whether characters near each end belong to the region, since the ends do not represent characters themselves, but rather character boundaries.

Figure 3.2 shows Epsilon with 3 windows. The top window and bottom window each show the buffer "main". Notice that although these two windows display the same buffer, they show different parts of the buffer. The mode line of the top window says 0%, but the mode line of the bottom window says 58%. The middle window displays a different buffer, named "other". If the cursor appears in the middle window and you type regular letters (the letters of your name, for example), they go into the buffer named "other" shown in that window. As you type the letters, the point (and so the cursor) stays to the right of the letters.
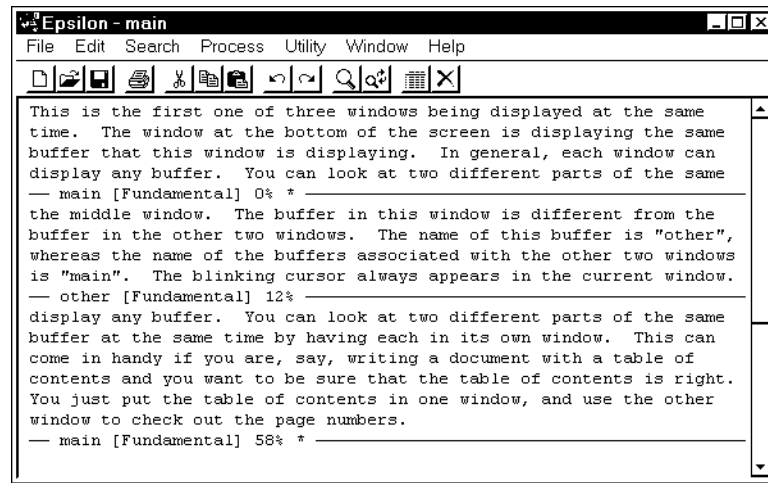
Figure 3.2: Epsilon with three windows.

In general, the *current window* refers to the window with the cursor, or the window where the "editing happens". The *current buffer* refers to the buffer displayed by the current window.

3.4   Different Keys for Different Uses: Modes

When you edit a C program, your editor should behave somewhat differently than when you write a letter, or edit a Lisp program, or edit some other kind of file.

For example, you might want the third function key to search forward for a comment in the current buffer. Naturally, what the editor should search for depends on the programming language in use. In fact, you might have Pascal in the top window and C in the bottom window.

To get the same key (in our example, the third function key) to do the right thing in either window, Epsilon allows each buffer to have its own interpretation of the keyboard.

We call such an interpretation a *mode*. Epsilon comes with several useful modes built in, and you can add your own using the Epsilon Extension Language (otherwise known as EEL, pronounced like the aquatic animal).

Epsilon uses the mode facility to provide the *dired* command, which stands for "directory edit". The dired command displays a directory listing in a buffer, and puts that buffer in dired mode. Whenever the current window displays that buffer, several special keys do things specific to dired mode. For example, the 'e' key displays the file listed on the current line of the directory listing, and the 'n' key moves down to the next line of the listing. See page 110 for a full description of dired mode.

Epsilon also provides C mode, which knows about several C indenting styles (see page 70). We also include Fundamental mode, a general-purpose editing mode.

The mode name that appears in a mode line suggests the keyboard interpretation active for the buffer displayed by that window. When you start Epsilon with no particular file to edit, Epsilon uses Fundamental Mode, so the word "Fundamental" appears in the mode line. Other words may appear after the mode name to signal changes, often changes particular to that buffer. We call these *minor modes*. For example, the auto-fill-mode command sets up a minor mode that automatically types a ⟨Return⟩ for you when you type near the end of a line. (See page 65.) It displays "Fill" in the mode line, after the name of the major mode. A read-only buffer display "RO" to indicate that you won't be able to modify it. There is always exactly one

major mode in effect for a buffer, but any number of minor modes may be active. Epsilon lists all active minor modes after the major mode's name.

3.5   Keystrokes and Commands: Bindings

Epsilon lets you redefine the function of nearly all the keys on the keyboard. We call the connection between a key and the command that runs when you type it a *binding*.

For example, when you type the ⟨Down⟩ key, Epsilon runs the down-line command. The down-line command, as the name suggests, moves the point down by one line. So when you type the ⟨Down⟩ key, point moves down by one line.

You can change a key's binding using the bind-to-key command. The command asks for the name of a command, and for a key. Thereafter, typing that key causes the indicated command to run. Using bind-to-key, you could, for example, configure Epsilon so that typing ⟨Down⟩ would run the forward-sentence command instead of the down-line command.

This key-binding mechanism provides a great deal of flexibility. Epsilon uses it even to handle the alphabetic and number keys that appear in the buffer when you type them. Most of the alphabetic and number keys run the command normal-character, which simply inserts the character that invoked it into the buffer.

Out of the box, Epsilon comes with a particular set of key bindings that make it resemble the EMACS text editor that runs on many kinds of computers. Using the key-binding mechanism and the bind-to-key command, you could rearrange the keyboard to make it resemble another editor's keyboard layout. That is exactly what the brief-keyboard command does; it rearranges the keyboard commands to make Epsilon work like the Brief text editor. See page 127.

Epsilon provides over 400 commands that you can bind to keys, and you can write brand new commands to do almost anything you want, and assign them to whatever keys you choose. See page 126 for more information on the bind-to-key command.

Some commands have no default binding. You can invoke any command, bound or not, by giving its name. The command named-command, normally bound to Alt-X, prompts for a command name and executes that command. For example, if you type

```
Alt-X down-line
```

followed by pressing the ⟨Enter⟩ key, the cursor moves down one line. Of course, you would find it easier in this example to simply type the ⟨Down⟩ key.

3.6   Repeating: Numeric Arguments

You can prefix a *numeric argument*, or simply an *argument*, to a command. This numeric argument generally functions as a repeat count for that command. You may enter a numeric argument in several ways. You may type Ctrl-U and then the number. You can also enter a numeric argument by holding down the Alt key and typing the number using the number keys across the *top* of the keyboard. Then you invoke a command, and that command generally repeats that number of times.

For example, suppose you type the four characters Ctrl-U 2 6 Ctrl-N. The Ctrl-N key runs the command named down-line, which moves point down one line. But given a numeric argument of 26, the command moves point down 26 lines instead of 1 line. If you give a numeric argument of -26 by typing a minus key while typing the 26, the down-line command would move point *up* 26 lines. You can get the same effect as Ctrl-U 2 6 Ctrl-N by holding down the Alt key and typing 26 on the main keyboard, then typing Ctrl-N.

(Remember to release the Alt key first; otherwise you'd get Alt-Ctrl-N.) For more information on numeric arguments, see page 22.

You can give a numeric argument to any Epsilon command. Most commands will repeat, as our example did above. But some commands use the numeric argument in some other way, which can vary from command to command. Some commands ignore the numeric argument. We describe all the commands in the chapter titled "Commands by Topic", which starts on page 31.

3.7 Viewing Lists

Sometimes Epsilon needs to show you a list of information. For example, when it asks you for the name of a file to edit, you might request a list of possible files to edit (see the next section). In such cases, Epsilon will display the list of items in a pop-up window. While in a pop-up window, one line will stand out in a different color. If you press ⟨Enter⟩, you select that item. To select another item, you can use normal Epsilon commands such as ⟨Up⟩ and ⟨Down⟩ to move to the next and previous items, or ⟨PageDown⟩ and ⟨PageUp⟩ to go to the next or previous windowful of items. You can even use Epsilon's searching commands to find the item you want. If you don't want any item on the list, you can simply type another response instead.

If you want to select one of the items and then edit it, press Alt-E. Epsilon will copy the highlighted line out of the list so can edit it.

3.8 Typing Less: Completion & Defaults

Whenever Epsilon asks you for some information (for instance, the name of a file you want to edit), you can use normal Epsilon commands to edit your response. For example, Control-A moves to the beginning of the response line. Most commands will work here, as long as the command itself doesn't need to prompt you for more information.

At many prompts, Epsilon will automatically type a default response for you, and highlight it. Editing the response will remove the highlight, while typing a new response will replace the default response. You can set the variable `insert-default-response` to zero if you don't want Epsilon to type in a response at prompts.

If you type a Control-R or Control-S, Epsilon will type in the default text. This is especially useful if you've told Epsilon not to automatically insert the default response, but it can also come in handy when you've mistakenly deleted or edited the default response, and you want to get it back. It's also convenient at prompts where Epsilon doesn't automatically type the default response, such as search prompts. Epsilon keeps separate defaults for the regular expression and non-regular expression replace commands, and for the regular expression and non-regular expression search commands. Epsilon will never overwrite what you actually type with a default, and indeed will only supply a default if you haven't yet specified any input for the response.

Another way to retrieve a previous response is to type Alt-E. While Ctrl-R and Ctrl-S provide a "suggested response" in many commands, Alt-E always types in exactly what you typed to that prompt last time. For example, at the prompt of the write-file command, Ctrl-S types in the name of the directory associated with the file shown in the current window, while Alt-E types in the last file name you typed at a write-file prompt. See page 26.

Alt-G provides yet another suggested response; it's often the name of the "current thing" for this prompt; in a search-and-replace command, for instance, Alt-G when typing the replacement text inserts the search text. In the write-file example, Alt-G inserts the current name of the file.

Sometimes Epsilon shows you the default in square brackets [ ]. This means that if you just press ⟨Enter⟩ without entering anything, Epsilon will use the value between the square brackets. You can also use
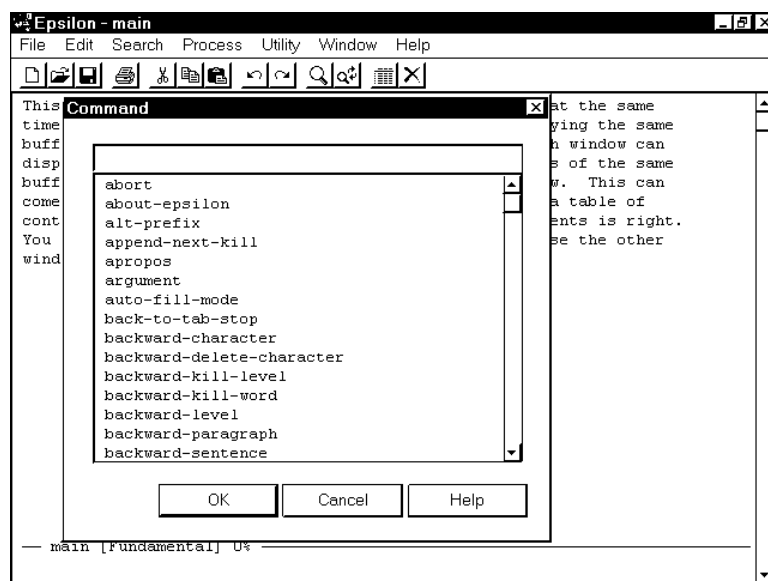
Figure 3.3: Typing '?' shows all of Epsilon's commands.

the Ctrl-S key here to pull in that default value, perhaps so that you can use regular Epsilon commands to edit the response string.

Epsilon can also retrieve text from the buffer at any prompt. Press the Alt-⟨Down⟩ key or Alt-Ctrl-N to grab the next word from the buffer and insert it in your response. Press the key again to retrieve successive words. This is handy if there's a file name in the buffer that you now want to edit, for example. The keys Alt-⟨PageDown⟩ or Alt-Ctrl-V behave similarly, but retrieve from the current position to the end of the line.

You can also use pull completion to retrieve text at a prompt that isn't at the current position, but elsewhere in the buffer. Begin typing the word you want to retrieve; then press Ctrl-⟨Up⟩ (or Ctrl-⟨Down⟩) to grab the previous (or next) word in the buffer that starts with what you've typed. F3 is the same as Ctrl-⟨Up⟩. See page 80 for details.

Whenever Epsilon asks for the name of something (like the name of a command, file, buffer, or tag), you can save keystrokes by performing *completion* on what you type. For example, suppose you type Alt-X to invoke a command by name, then type the letter 'v'. Only one command begins with the letter 'v', the visit-file command. Epsilon determines that you mean the visit-file command by examining its list of commands, and fills in the rest of the name. We call this process *completion*.

To use completion, type a ⟨Space⟩ and Epsilon will fill in as much of the name as possible. The letters Epsilon adds will appear as if you had typed them yourself. You can enter them by typing ⟨Enter⟩, edit them with normal editing commands, or add more letters. If Epsilon cannot add any letters when you ask for completion, it will pop up a list of items that match what you've typed so far. To disable automatic pop-ups on completion, set the `completion-pops-up` variable to zero.

For example, four commands begin with the letters "go", goto-beginning, goto-end, goto-line, and goto-tag. If you type "go", and then press ⟨Space⟩, Epsilon fills in "goto-" and waits for you to type more. Type 'b' and another ⟨Space⟩, to see "goto-beginning". Epsilon moves the cursor one space to the right of the last letter, to indicate a match. Press ⟨Enter⟩ to execute the goto-beginning command.

The ⟨Esc⟩ key works just like the ⟨Space⟩ key, except that if a single match results from the completion, Epsilon takes that as your response. This saves you a keystroke, but you don't have the opportunity to check the name before continuing. The ⟨Tab⟩ key does the same thing. However, inside a dialog under Windows,
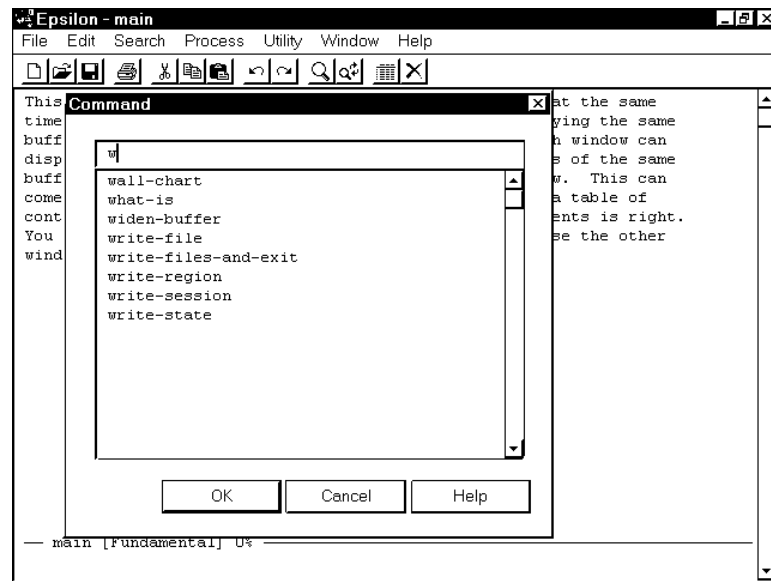
Figure 3.4: Typing "w?" shows all commands that start with 'w'.

these two keys perform their usual Windows functions of canceling the dialog, and moving around in the dialog, respectively. They aren't used for completion.

Typing a question mark during completion causes Epsilon to display a list of choices in a pop-up window. Recall that completion works with buffer and file names, as well as with command names. For example, you can get a quick directory listing by giving any file command and typing a question mark when asked for the file name. Press the Ctrl-G key to abort the command, when you've read the listing. (See the dired command on page 110 for a more general facility.)

Figure 3.3 shows you what Epsilon looks like when you type Alt-X (the named-command command), and then press '?' to see a list of the possible commands. Epsilon shows you all its commands in a pop-up window. Epsilon provides many more commands than could fit in the window, so Epsilon shows you the first window-full. At this point, you could press ⟨Space⟩ or ⟨PgDn⟩ to see the next window-full of commands, or use searching or other Epsilon commands to go to the item you desire. If you want the highlighted item, simply press ⟨Enter⟩ to accept it. If you type Alt-E, Epsilon types in the current item and allows you to edit it. Type any normal character to leave the pop-up window and begin entering a response by hand.

Figure 3.4 shows what the screen looks like if you type 'w' after the Alt-X, then type '?' to see the list of possible completions. Epsilon lists the commands that start with 'w'.

You can set variables to alter Epsilon's behavior. The menu-width variable contains the width of the pop-up window of matches that Epsilon creates when you press '?'. (Unix only. In Windows, drag the dialog's border to change its size.) The search-in-menu variable controls what Epsilon does when you press '?' and then continue typing a response. If it has a value of zero, as it does by default, Epsilon moves from the pop-up window back to the prompt at the bottom of the screen, and editing keys like ⟨Left⟩ navigate in the response. If search-in-menu has a nonzero value, Epsilon moves in the pop-up menu of names to the first name that matches what you've typed, and stays in the pop-up window. (If it can't find a match, Epsilon moves back to the prompt as before.)

During file name completion, Epsilon can ignore files with certain extensions. The ignore-file-extensions variable contains a list of extensions to ignore. By default, this variable has the value '|.obj|.exe|.o|.b|', which makes file completion ignore files that end with .obj, .exe, .o, and .b.

Each extension must appear between '|' characters. You can augment this list using the set-variable command, described on page 127.

Similarly, the `only-file-extensions` variable makes completion look only for files with certain extensions. It uses the same format as `ignore-file-extensions`, a list of extensions surrounded by | characters. If the variable holds a null pointer, Epsilon uses `ignore-file-extensions` as above.

3.9   Command History

Epsilon maintains a list of your previous responses to all prompts. To select a prompt from the list, press the Alt-⟨Up⟩ key or Alt-Ctrl-P. Then use the arrow keys or the mouse to choose a previous response, and press ⟨Enter⟩. If you want to edit the response first, press Alt-E.

For example, when you use the grep command to search in files for a pattern, you can press Alt-⟨Up⟩ to see a list of file patterns you've used before. If the pattern `\windows\system\*.inf` appeared on the list, you could position the cursor on it and then press Alt-E. Epsilon would copy the pattern out of the list so you can edit it, perhaps replacing `*.inf` with `*.ini`. Both patterns would then appear in the history list next time. Or you could just press ⟨Enter⟩ in the list of previous responses to use the same pattern.

You can also use Alt-E at any prompt to retrieve the last response without showing a list of responses first. For example, Ctrl-X Ctrl-F Alt-E will insert the full name of the last file you edited with the find-file command.

Except in certain searching commands, you can ⟨Up⟩ or Ctrl-P instead of Alt-⟨Up⟩ key or Alt-Ctrl-P. These normally behave the same, but you can set the `recall-prior-response-options` variable to make the non-Alt versions of the keys select older command history responses without displaying a list of all of them.

3.10   Mouse Support

Epsilon supports a mouse under Windows and under X in Unix. You can use the left button to position point, or drag to select text. Double-clicking selects full words. (When a pop-up list of choices appears on the screen, double-clicking on a choice selects it.) Use shift-clicking to extend or contract the current selection by repositioning the end of the selection. Holding down the Alt key while selecting produces a rectangle selection.

Once you've selected a highlighted region, you can drag it to another part of the buffer. Move the mouse inside the highlighted region, hold down a mouse button and move the mouse to another part of the buffer while holding down the button. The mouse cursor changes to indicate that you're dragging text. Release the mouse button and the text will move to the new location. To make a copy of the text instead of moving it, hold down the Control key while dropping the text.

Dragging text with the mouse also copies the text to a kill buffer, just as if you had used the corresponding keyboard commands to kill the text and yank it somewhere else. When you drag a highlighted rectangular region of text, Epsilon's behavior depends upon the whether or not the buffer is in overwrite mode. In overwrite mode, Epsilon removes the text from its original location, replacing it with spaces. Then it puts the text in its new location, overwriting whatever text might be there before. In insert mode, Epsilon removes the text from its original location and shifts text to its right leftwards to fill the space it occupied. Then it shifts text to the right in the new location, making room for the text.

You can use the left button to resize windows by dragging window corners or borders. For pop-up windows only, dragging the title bar moves the window.

A pop-up window usually has a scroll bar on its right border. Drag the box or diamond up and down to scroll the window. Click on the arrows at the top or bottom to scroll by one line. Click elsewhere in the

scroll bar to scroll by a page. In some environments, ordinary tiled windows have a scroll bar that pops up when you move the mouse over the window's right-hand border, or (for windows that extend to the right edge of the screen), when you move the mouse past the right edge. The toggle-scroll-bar command toggles whether tiled windows have pop-up scroll bars or permanent scroll bars.

Under X, you can adjust the speed at which Epsilon scrolls due to mouse movements by setting the `scroll-rate` variable. It contains the number of lines to scroll per second. The `scroll-init-delay` variable contains the delay in hundredths of a second from the time the mouse button goes down and Epsilon scrolls the first time, to the time Epsilon begins scrolling repeatedly.

In Epsilon for Windows, the right button displays a context menu (which you can modify by editing the file gui.mnu). In other versions, the right mouse button acts much like the left button, but with a few differences: On window borders, the right button always resizes windows, rather than scrolling or moving them. When you double-click with the right mouse button on a subroutine name in a buffer in C mode, Epsilon goes to the definition of that subroutine using the pluck-tag command (see page 43). To turn off this behavior in a particular buffer, set the buffer-specific variable `mouse-goes-to-tag` to zero. To make the right button jump to a subroutine's definition when you double-click in any buffer, not just C mode buffers, set the default value of this variable to one. If you don't want C mode to automatically set this variable nonzero, set the variable `c-mode-mouse-to-tag` to zero.

You can click (or hold) the middle mouse button and drag the mouse to pan or auto-scroll—the speed and direction of scrolling varies as you move the mouse. This works on wheeled mice or on any mouse with three buttons. See the `mouse-center-yanks` variable to make the center button yank text instead, or do nothing.

Epsilon for Windows or Unix (under X) also recognizes wheel rolling on wheeled mice, and scrolls the current window when you roll the wheel. See the `wheel-click-lines` variable for more details.

In Epsilon for Unix, selecting text normally copies it to the clipboard. See the variable `mouse-selection-copies` to change this behavior.

3.11   The Menu Bar

The Windows GUI version of Epsilon provides a customizable menu bar and tool bar. To modify the menu bar, edit the file gui.mnu. Comments in the file describe its format. To modify the tool bar, you can redefine the EEL command standard-toolbar in the file menu.e.

Other versions of Epsilon provide a text-based menu bar, which is hidden by default. Most of the customization variables described below only apply to the text-based menu bar.

You can have Epsilon display a menu bar all the time with the toggle-menu-bar command, or press Alt-F2 (the show-menu command) to display it at any time, and hide it again after you select a command. When you use the menu bar to invoke a command that needs additional input, Epsilon automatically brings up a list of options (as if you typed '?') so that you can select one without using the keyboard.

You can change the contents of the menu bar by editing the file epsilon.mnu. Comments in the file describe its format. Epsilon stores the name of its menu file in the variable `menu-file`. Set this variable to make Epsilon use a different menu file. During Brief emulation, Epsilon uses the menu file brief.mnu. Epsilon for Windows uses the variable `gui-menu-file` instead, and the file `gui.mnu` by default.

If you hold down the Shift or Ctrl keys while selecting a menu bar command, Epsilon will run the command with a numeric argument of 1. This is handy for commands that behave differently when given a numeric argument. When you select an item on the text-based menu bar, Epsilon flashes the selected item. The `menu-bar-flashes` variable holds the number of flashes (default two).

By default, Epsilon displays key bindings for menu items. Set the variable `menu-bindings` to zero to disable this feature. (Epsilon for Windows ignores this variable and always displays such bindings.) Epsilon

```
  File  Edit  Window  Search  Process  Utility  Help
              Select buffer...      Ctrl-X Ctrl-B
#             One window            Ctrl-X 1
# ->update_ca Kill window           C-X C-D
#             Zoom window           Ctrl-X z
# Recompute a                                    curate polar form.
#             Horizontal split      Ctrl-X 2
sub update_ca Vertical split        Ctrl-X 5
       my $s
       my ($  Set colors...
       $self  Set display look...      ⤡
       retur  Toggle borders                  os($t), $r * CORE::sin($t)];
}             Show spaces           Shift-F6
              Change line wrapping
#
#             Toggle menu bar
# ->update_po Toggle scroll bars
#
# Recompute and return the polar form, given accurate cartesian form.
#
sub update_polar {
       my $self = shift;
       my ($x, $y) = @{$self->{'cartesian'}};
       $self->{p_dirty} = 0;
       return $self->{'polar'} = [0, 0] if $x == 0 && $y == 0;
       return $self->{'polar'} = [CORE::sqrt($x*$x + $y*$y),
                                  CORE::atan2($y, $x)];
— complex.pm [Perl Fill] 12% update_polar ———————
```

Figure 3.5: Epsilon's text-based menu bar.

computes bindings dynamically the first time it displays a particular menu column. (For several commands with multiple bindings, the epsilon.mnu file selects a particular binding to display.) The rebuild-menu command makes Epsilon reconstruct its menus: use this command after setting menu-bindings.

By default, when you click on the text-based menu bar but release the mouse without selecting a command, Epsilon leaves the menu displayed until you click again. Set the `menu-stays-after-click` variable to zero if you want Epsilon to remove the menu when this happens.

Chapter 4

Commands by Topic

This chapter lists all the Epsilon commands, grouped by topic. Each section ends with a summary of the keys, and the names you would use to invoke the commands by name, or to rebind them to other keys.

4.1   Getting Help

You can get help on Epsilon by typing F1, the *help key*. The help key will provide help at any time. If you type it during another command, help simply pops up a description of that command. Otherwise, the help command asks you to type an additional key to indicate what sort of help you want. Many of these options are also available directly from Epsilon's Help menu item, in versions with a menu bar.

The help command actually uses various commands which you can invoke individually. Here are the keys you can use at the help prompt.

Pressing **A** invokes the apropos command, which asks for a string, looks through the short descriptions of all the commands and variables, then pops up a list of commands or variables (and their descriptions) that contain the string, along with their key bindings. Highlighted words are links to the full documentation.

Help's **K** option invokes the describe-key command. It prompts for a key and provides full documentation on what that key does.

The **C** option invokes the command describe-command, which provides full documentation on the command whose name you specify, and also tells which keys invoke that command.

The **B** option invokes the command show-bindings, which asks for a command name and gives you the keys that run that command.

The **I** option invokes the command info, which starts Info mode. Info mode lets you read the entire Epsilon manual, as well as any other documentation you may have in Info format. See page 32.

The **F** option is a shortcut into Epsilon's manual in Info mode. It prompts for some text, then looks up that text in the index of Epsilon's online manual. Just press ⟨Enter⟩ to go to the top of the manual. This option invokes the command epsilon-info-look-up; the command epsilon-manual-info goes to the top of Epsilon's documentation without prompting.

The **Ctrl-C** option prompts for the name of an Epsilon command, then displays an Info page from Epsilon's online manual that describes the command.

The **Ctrl-K** option prompts for a key, then displays an Info page from Epsilon's online manual that describes the command it runs.

The **Ctrl-V** option prompts for an Epsilon variable's name, then displays an Info page from Epsilon's online manual that describes that variable.

The **H** option displays Epsilon's manual in HTML format, by running a web browser. It prompts for a topic, which can be a command or variable name, or any other text. (The browser will try to find an exact match for what you type; if not, it will search for web pages containing that word.) When you're looking at Epsilon's manual in Info mode, using one of the previous commands, this command will default to showing the same topic in a browser.

The **W** option, in Epsilon for Windows, displays Epsilon's WinHelp help file. Like the Info-format manual, it contains the complete text of the Epsilon manual.

The **Q** option invokes the command what-is, which asks for a key and tells you what command would run if you typed that key.

The **R** option invokes the describe-variable command, which asks for a variable name and displays the help on that variable.

The **L** option invokes the show-last-keys command, which pops up a window that displays the last 60 keystrokes you typed.

The **M** option displays help on the major mode of the current buffer. For example, when you're editing a C file, this command displays help on C mode.

The **V** command displays Epsilon's version number and similar information.

The **?** option displays information on the help command itself, including its options, just as typing the help key again would.

The **B** and **Q** options tell you about bindings without showing you the associated documentation on the command. In contrast to the first three options, these two display their information in the echo area, instead of popping up a window.

The wall-chart command creates a table showing the commands invoked by all the keys. It builds a chart in a buffer named "wall". The wall chart includes any changes you may have made to the normal key bindings. You can print it and attach it to any convenient wall using the print-buffer command.

Epsilon's help system keeps track of any changes that you make to Epsilon. For example, if you completely remap the keyboard, Epsilon's help system will know about it and still give you correct key binding information. And Epsilon's help system will also keep track of any commands or keyboard macros that you write and add to Epsilon.

The release-notes command reads and displays the release notes for this version of Epsilon.

Some of Epsilon's help commands use the on-line documentation file, edoc. This file contains descriptions for each of Epsilon's commands and variables. See the description of the −fd flag on page 12.

Under Windows, help commands normally use the standard Windows help file edoc.hlp to provide help. This file contains the complete Epsilon manual. You can set the variable `want-gui-help` to zero if you want Epsilon to use its built-in help system instead of Windows-style help whenever it can; you'll still be able to use the epsilon-manual command to get Windows-style help. The Win32 console version of Epsilon uses a similar variable `want-gui-help-console`.

Summary:	F1, Alt-?, Ctrl-_	help
	F1 A	apropos
	F1 K	describe-key
	F1 C	describe-command
	F1 R	describe-variable
	F1 L	show-last-keys
	F1 Q, F6	what-is
	F1 B, F5	show-bindings
	F1 Ctrl-C	info-goto-epsilon-command
	F1 Ctrl-K	info-goto-epsilon-key
	F1 Ctrl-V	info-goto-epsilon-variable
	F1 V	about-epsilon
	F1 F	epsilon-info-look-up
		wall-chart
		release-notes
		epsilon-manual
		epsilon-manual-info

4.1.1   Info Mode

Epsilon's Info mode lets you read documentation in Info format. You can press F1 i to start Info mode. One example of documentation available in Info format is Epsilon's manual.

An Info document is divided into nodes. Each node describes a specific topic. Nodes are normally linked together into a tree structure.

Every node has a name, which appears on the very first line of the node. The first line might look like this:

```
File: cp,  Node: Files,  Next: More Options,  Prev: Flags,  Up: Top
```

That line also indicates that the node named "More Options" comes next after this "Files" node. And it says which node comes before it, and which node is its parent. (Some nodes don't have a "Next" or a "Prev" or an "Up" node.) In Info mode, the keys N, P, and U move to the current node's Next node, its Prev node, or its Up node (its parent node).

You can scroll through a node with the usual Epsilon commands, but Info mode also lets you use ⟨Space⟩ to page forward and ⟨Backspace⟩ to page back. When you're at the end of a node, the ⟨Space⟩ key goes on to the next one, walking the tree structure so you can read through an entire Info file. The ⟨Backspace⟩ key does the reverse; it goes to the previous node when you press it and you're already at the top of a node. (The keys] and [move ahead and back similarly, but don't page; use them when you don't want to see any more of the current node.)

Some nodes have menus. They look like this:

```
* Menu:

* Buffers::
* Flags::
* Switches: Flags.
```

Press the M key to select an item from a menu, then type the name of the item (the part before the : character). You can press ⟨Space⟩ to complete the name, or type just part of the name. The first two menu items let you type Buffers or Flags and go to a node with that same name; the last item lets you type Switches but Epsilon will go to a node named Flags.

You can also press a digit like 1, 2, 3 to go to the corresponding node in the current node's menu. Press 0 to go to the last node, whatever its number. So in the menu above, either 3 or 0 would go to the Flags node. Typically when you select a node from a menu, that node's Up will lead back to the node with the menu.

A node can also have cross-references. A cross-reference looks like this: *Note: Command History::. Use the F key to follow a cross reference; it completes like M does.

Instead of typing M or F followed by a node name, you can use ⟨Tab⟩ and ⟨Backtab⟩ to move around in a node to the next or previous menu item or cross-reference, then press ⟨Enter⟩ to follow it. Or you can double-click with the mouse to follow one.

Epsilon keeps a history of the Info nodes you've visited, so you can retrace your steps. Press L to go to the last Info node you were at before this one. Press L repeatedly to revisit earlier nodes. When you're done looking at Info documentation, press Q to exit Info mode.

Info documentation is tree-structured. Normally each separate program has its own file of documentation, and the nodes within form a tree. Each Info file normally has a node named "top" that's the top node in its tree. Then all the trees are linked together in a directory file named "dir", which contains a menu listing all the available files. The T key goes to the top node in the current file. The D key goes to the top node in the directory file.

When a node name reference contains a word in parentheses, like (epsilon)Language Modes, it indicates the node is in a file whose name is inside the parentheses. (Otherwise the node must be in the current file.) If you omit the node name and just say (epsilon), the Top node is implied.

When a complete path to an Info file isn't specified (as is usually the case), Epsilon looks along an Info path. First it looks in each directory of the colon-separated list in the variable `info-path-unix` (or, in non-Unix versions of Epsilon, the semicolon-separated list in `info-path-non-unix`). These paths may use `%x` to indicate the directory containing Epsilon's executable. If the Info file still isn't found, Epsilon tries directories listed in any INFOPATH environment variable.

Press S to search in an Info file. You can use the same keys as in other Epsilon search commands to perform a regular expression search, word search, or control case folding. This command will jump from node to node if necessary to find the next match. If you use normal searching keys like Ctrl-S or Ctrl-R, they will report a failing search if there are no more matches in the current node. Press Ctrl-S or Ctrl-R again to have Epsilon continue the search into other nodes.

Press I to use an Info file's index. I ⟨Enter⟩ simply moves to the first index node in a file. Or you can type some text, and Epsilon will display each of the nodes in the file that have an index entry containing that text. Use ⟨Comma⟩ to advance to the next such entry.

There are a few more Info commands. B goes to the beginning of the current node, like Alt-<. > goes to the last node of the file, viewed as a hierarchy. G prompts for the name of a node, then goes there. (You can use it to reach files that might not be linked into the Info hierarchy.) H displays this documentation. And ? displays a short list of Info commands.

You can navigate to Epsilon's manual using Info commands, as explained above, but Epsilon also provides some shortcut commands. Press F1 Ctrl-C to look up an Epsilon command's full documentation by command name. Press F1 Ctrl-K, then press any key and Epsilon will show the documentation for whatever command it runs. Press F1 Ctrl-V to look up a variable. Press F1 f ⟨Enter⟩ to go to the top of Epsilon's documentation tree, or type a topic name before the ⟨Enter⟩ and Epsilon will look up that word in the index to Epsilon's online documentation.

If you write you own Info file, Epsilon provides some commands that help. The info-validate command checks an Info file for errors (such as using a nonexistent node name). The info-tagify command builds or updates an Info file's tag table. (Info readers like Epsilon can find nodes more quickly when a file's tag table is up to date, so run this after you modify an Info file.)

Summary:	Info mode only: N	info-next
	Info mode only: P	info-previous
	Info mode only: U	info-up
	Info mode only: ⟨Space⟩	info-next-page
	Info mode only: ⟨Backspace⟩	info-previous-page
	Info mode only: [	info-backward-node
	Info mode only:]	info-forward-node
	Info mode only: M	info-menu
	Info mode only: 0, 1, 2, ...	info-nth-menu-item
	Info mode only: F	info-follow-reference
	Info mode only: ⟨Tab⟩	info-next-reference
	Info mode only: Shift-⟨Tab⟩	info-previous-reference
	Info mode only: ⟨Enter⟩	info-follow-nearest-reference
	Info mode only: L	info-last
	Info mode only: Q	info-quit
	Info mode only: T	info-top
	Info mode only: D	info-directory-node
	Info mode only: S	info-search
	Info mode only: I	info-index

Info mode only: ⟨Comma⟩	info-index-next
Info mode only: >	info-last-node
Info mode only: G	info-goto
	info
	info-mode
	info-validate
	info-tagify

4.1.2 Web-based Epsilon Documentation

Epsilon's online manual is available in three formats:

- You can read the manual in an Epsilon buffer using Info mode by pressing F1 f. See page 32.

- Users running Microsoft Windows can access the WinHelp version of the manual by pressing F1 w. See page 31 for more information.

- You can view the HTML version of the manual using a web browser by pressing F1 h.

To display the HTML manual, Epsilon starts a documentation server program. This is named lhelp.exe (or lhelpd in Unix). The documentation server runs in the background, hiding itself from view, and your web browser communicates with it on a special "port", as if it were a web server.

The documentation server must be running in order to serve documentation, so a bookmark to a page in the documentation will only work if the documentation server is running. You can press F1 h in Epsilon to ensure it's running. To force an instance of the documentation server to exit, invoke it again with the -q flag.

If your browser is configured to use a proxy, you will typically need to tell it not to use proxy settings for addresses starting with 127.0.0.1 so that it may connect to the local documentation server.

Epsilon for Unix uses a shell script named `goto_url` to run a browser. You can edit it if you prefer a different browser. Epsilon will first look for a customized copy of `goto_url` in your `~/.epsilon` directory. If there is none, it will search for and invoke any customized copy of `goto_url` it finds on your path. Failing that, it will use the standard copy installed in Epsilon's bin directory. Epsilon for Windows uses the system's default browser.

4.2 Moving Around

4.2.1 Simple Movement Commands

The most basic commands involve moving point around. Recall from page 20 that point refers to the place where editing happens.

The Ctrl-F command moves point forward one character, and Ctrl-B moves it back. Ctrl-A moves to the beginning of the line, and Ctrl-E moves to its end.

Ctrl-N and Ctrl-P move point to the next and previous lines, respectively. They will try to stay in the same column in the new line, but will never expand a line in order to maintain the column; instead they will move to the end of the line (but see below). The key Alt-< moves point before the first character in the buffer, and Alt-> moves point after the last character in the buffer.

You can use the arrow keys if you prefer: the ⟨Right⟩ key moves forward a character, ⟨Left⟩ moves back a character, ⟨Down⟩ moves down a line, and ⟨Up⟩ moves up a line. Most commands bound to keys on the

numeric keypad also have bindings on some control or alt key for those who prefer not to use the keypad. Throughout the rest of this chapter, the explanatory text will only mention one of the bindings in such cases; the other bindings will appear in the summary at the end of each section.

By default, pressing ⟨Right⟩ at the end of the line moves to the start of the next line. When you press ⟨Down⟩ at the end of a 60-character line, and the next line only has 10 characters, Epsilon moves the cursor back to column 10. You can change this by setting the buffer-specific `virtual-space` variable (by default zero). If you set it to one, the ⟨Up⟩ and ⟨Down⟩ keys will stay in the same column, even if no text exists there. If you set it to two, in addition to ⟨Up⟩ and ⟨Down⟩, the ⟨Right⟩ and ⟨Left⟩ keys will move into places where no text exists. Setting `virtual-space` to two only works correctly on lines longer than the window when Epsilon has been set to scroll long lines (the default), rather than wrapping them (see page 85). Some commands behave unexpectedly on wrapped lines when `virtual-space` is two.

When you move past the bottom or top of the screen using ⟨Up⟩ or ⟨Down⟩, Epsilon scrolls the window by one line, so that point remains at the edge of the window. If you set the variable `scroll-at-end` (normally 1) to a positive number, Epsilon will scroll by that many lines when ⟨Up⟩ or ⟨Down⟩ would leave the window. Set the variable to 0 if you want Epsilon to instead center the current line in the window.

Summary: Ctrl-A, Alt-⟨Left⟩   beginning-of-line
               Ctrl-E, Alt-⟨Right⟩   end-of-line
               Ctrl-N, ⟨Down⟩   down-line
               Ctrl-P, ⟨Up⟩   up-line
               Ctrl-F, ⟨Right⟩   forward-character
               Ctrl-B, ⟨Left⟩   backward-character
               Alt-<, Ctrl-⟨Home⟩   goto-beginning
               Alt->, Ctrl-⟨End⟩   goto-end

4.2.2   Moving in Larger Units

Words

Epsilon has several commands that operate on words. A word usually consists of a sequence of letters, numbers, and underscores. The Alt-F and Alt-B commands move forward and backward by words, and the Alt-D and Alt-⟨Backspace⟩ commands kill forward and backward by words, respectively. Like all killing commands, they save away what they erase (see page 49 for a discussion on the killing commands). Epsilon's word commands work by moving in the appropriate direction until they encounter a word edge.

The word commands use a regular expression to define the current notion of a word. They use the buffer-specific variable `word-pattern`. This allows different modes to have different notions of what constitutes a word. Most built-in modes, however, make `word-pattern` refer to the variable `default-word`, which you can modify. See page 55 for information on regular expressions, and page 127 for information on setting this variable.

You can set the `forward-word-to-start` variable nonzero if you want Epsilon to stop at the start of a word instead of at its end when moving forward.

Summary: Alt-F, Ctrl-⟨Right⟩   forward-word
               Alt-B, Ctrl-⟨Left⟩   backward-word
               Alt-⟨Backspace⟩   backward-kill-word
               Alt-D   kill-word

Sentences

For sentences, Epsilon has the Alt-E and Alt-A keys, which move forward and backward by sentences, and the Alt-K key, which deletes forward to the end of the current sentence. A sentence ends with one of the characters period, !, or ?, followed by any number of the characters ", ',),], followed by two spaces or a newline. A sentence also ends at the end of a paragraph. The next section describes Epsilon's notion of a paragraph.

You can set the `sentence-end-double-space` variable to change Epsilon's notion of a sentence. The commands in this section will require only one space at the end of a sentence, and paragraph filling commands will use one space as well. Note that Epsilon won't be able to distinguish abbreviations from the ends of sentences with this style.

Summary: Alt-E   forward-sentence
             Alt-A   backward-sentence
             Alt-K   kill-sentence

Paragraphs

For paragraphs, the keys Alt-] and Alt-[move forward and back, and the key Alt-H puts point and mark around the current paragraph. Blank lines (containing only spaces and tabs) always separate paragraphs, and so does the form-feed character ^L.

You can control what Epsilon considers a paragraph using two variables.

If the buffer-specific variable `indents-separate-paragraphs` has a nonzero value, then a paragraph also begins with a nonblank line that starts with a tab or a space.

If the buffer-specific variable `tex-paragraphs` has a nonzero value, then Epsilon will not consider as part of a paragraph any sequence of lines that each start with at sign or period, if that sequence appears next to a blank line. And lines starting with \begin or \end, or with %, \[, \], or $$, will also delimit paragraphs.

Summary: Alt-], Alt-⟨Down⟩   forward-paragraph
             Alt-[, Alt-⟨Up⟩   backward-paragraph
             Alt-H   mark-paragraph

Parenthetic Expressions

Epsilon has commands to deal with matching parentheses, square brackets, curly braces, and similar delimiters. We call a pair of these characters with text between them a *level*. You can use these level commands to manipulate expressions in many programming languages, such as Lisp, C, and Epsilon's own embedded programming language, EEL.

A level can contain other levels, and Epsilon won't get confused by the inner levels. For example, in the text "one (two (three) four) five" the string "(two (three) four)" constitutes a level. Epsilon recognizes that "(three)" also constitutes a level, and so avoids the mistake of perhaps calling "(two (three)" a level. In each level, the text inside the delimiters must contain matched pairs of that delimiter. In many modes, Epsilon knows to ignore delimiters inside strings or comments, when appropriate.

Epsilon typically recognizes the following pairs of enclosures: '(' and ')', '[' and ']', '{' and '}'. The command Ctrl-Alt-F moves forward to the end of the next level, by looking forward until it sees the start of

a level, and moving to its end. The command Ctrl-Alt-B moves backward by looking back for the end of a level and going to its beginning. The Ctrl-Alt-K command kills the next level by moving over text like Ctrl-Alt-F and killing as it travels, and the Alt-⟨Del⟩ command moves backward like Ctrl-Alt-B and kills as it travels. A mode may define a different set of grouping characters, such as < and > for HTML mode.

The Alt-) key runs the find-delimiter command. Use it to temporarily display a matching delimiter. The command moves backward like Ctrl-Alt-B and pauses for a moment, showing the screen, then restores the screen as before. The pause normally lasts one half of a second, or one second if the command must temporarily reposition the window to show the matching delimiter. You can specify the number of hundredths of a second to pause by setting the variables `near-pause` and `far-pause`. Also, typing any key will immediately restore the original window context, without further pause.

The show-matching-delimiter command inserts the key that invoked it by calling normal-character and then invokes find-delimiter to show its match. The maybe-show-matching-delimiter command is similar, but only invokes find-delimiter if the `Matchdelim` variable is nonzero. In Fundamental mode, the ')', ']' and '}' keys run maybe-show-matching-delimiter.

In some modes, when the cursor is over a delimiter Epsilon will automatically seek out its matching delimiter and highlight them both. See the descriptions of the individual modes for more information.

Summary:	Alt-)	find-delimiter
	Ctrl-Alt-F	forward-level
	Ctrl-Alt-B	backward-level
	Ctrl-Alt-K	kill-level
	Alt-⟨Del⟩	backward-kill-level
		show-matching-delimiter

4.2.3 Searching

Epsilon provides a set of flexible searching commands that incorporate *incremental search*. In the incremental-search command, Epsilon searches as you type the search string. Ctrl-S begins an incremental search forward, and Ctrl-R starts one in reverse. Any character that normally inserts itself into the buffer becomes part of the search string. In an incremental search, Ctrl-S and Ctrl-R find the next occurrence of the string in the forward and reverse directions, respectively. With an empty search string, Ctrl-S or Ctrl-R will either reverse the direction of the search, or bring in the previously used search string. (To retrieve older search strings, see page 26.)

You can use ⟨Backspace⟩ to remove characters from the search string, and enter control characters and *meta characters* (characters with the eighth bit set) in the search string by quoting them with Ctrl-Q. (Type Ctrl-Q Ctrl-J to search for a ⟨Newline⟩ character.) Use the Ctrl-G abort command to stop a long search in progress.

Typing ⟨Enter⟩ or ⟨Esc⟩ exits from an incremental search, makes Epsilon remember the search string, and leaves point at the match in the buffer.

While typing characters into the search string for incremental-search, a Ctrl-G quits and moves point back to the place the search started, without changing the default search string. During a failing search, however, Ctrl-G simply removes the part of the string that did not match.

If you type an editing key not mentioned in this section, Epsilon exits the incremental search, then executes the command bound to the key.

You can make Epsilon copy search text from the current buffer by typing Alt-⟨Down⟩. Epsilon will append the next word from the buffer to the current search string. This is especially convenient when you see

a long variable name, and you want to search for other references to it. (It's similar to setting the mark and moving forward one word with Alt-F, then copying the text to a kill buffer and yanking it into the current search string.) Similarly, Alt-⟨PageDown⟩ appends the next line from the current buffer to the search string. These two keys are actually available at almost any Epsilon prompt, though they're especially useful when searching. Alt-Ctrl-N and Alt-Ctrl-V are synonyms for Alt-⟨Down⟩ and Alt-⟨PageDown⟩, respectively.

While Alt-⟨Down⟩ and Alt-⟨PageDown⟩ copy text from the buffer at point, using the word pulling keys F3, Ctrl-⟨Up⟩ or Ctrl-⟨Down⟩ copies text into the search string from other parts of the buffer; see page 80.

You can change how Epsilon interprets the search string by pressing certain keys when you type in the search string. Pressing the key a second time restores the original interpretation of the search string.

- Pressing Ctrl-C toggles the state of *case folding*. While case folding, Epsilon considers upper case and lower case the same when searching, so a search string of "Word" would match "word" and "WORD" as well.

  Epsilon remembers the state of case folding for each buffer separately, using the buffer-specific variable `case-fold`. When you start to search, Epsilon sets its default for case folding based on that variable's value for the current buffer. Toggling case folding with Ctrl-C won't affect the default. Use the `set-variable` command described on page 127 to do this.

- Pressing Ctrl-W toggles *word searching*. During word searching, Epsilon only looks for matches consisting of complete words. For instance, word searching for 'a' in this paragraph finds only one match (the one in quotes), but eleven when not doing word searching. You can type multiple words separated by spaces, and Epsilon will recognize them no matter what whitespace characters separate them (for instance, if they're on successive lines).

- Pressing Ctrl-T makes Epsilon interpret the search string as a regular expression search pattern, as described on page 55. Another Ctrl-T turns off this interpretation. If the current search string denotes an invalid regular expression, Epsilon displays "Bad R-E Search: `<string>`" instead of its usual message "R-E Search: `<string>`" (where `<string>` refers to the search string).

- Pressing Ctrl-O toggles incremental searching. In an incremental search, most editing commands will exit the search, as described above. But you may want to edit the search string itself. If you turn off the "incremental" part of incremental search with the Ctrl-O key, Epsilon will let you use the normal editing keys to modify the search string.

  In non-incremental mode, Epsilon won't automatically search after you type each character, but you can tell it to find the next match by typing Ctrl-S or Ctrl-R (depending on the direction). This performs the search but leaves you in search mode, so you can find the next occurrence of the search string by typing Ctrl-S or Ctrl-R again. When you press ⟨Enter⟩ to exit from the search, Epsilon will search for the string you've entered, unless you've just searched with Ctrl-S or Ctrl-R. (In general, the ⟨Enter⟩ key causes a search if the cursor appears in the echo area. If, on the other hand, the cursor appears in a window showing you a successful search, then typing the ⟨Enter⟩ key simply stops the search.) A numeric argument of *n* to a non-incremental search will force Epsilon to find the *n*th occurrence of the indicated string.

Epsilon interprets the first character you type after starting a search with Ctrl-S or Ctrl-R a little differently. Normally, Ctrl-S starts an incremental search, with regular expression searching and word searching both disabled. If you type Ctrl-T or Ctrl-W to turn one of these modes on, Epsilon will also turn off incremental searching. Epsilon also pulls in a default search string differently if you do it immediately. It will always provide the search string from the last search, interpreting the string as it did for that search. If you retrieve a default search string at any other time, Epsilon will provide the last one consistent with the state of regular expression mode (in other words, the last regular expression pattern, if in regular expression mode, or the last non-regular-expression string otherwise).

The Ctrl-Alt-S and Ctrl-Alt-R commands function like Ctrl-S and Ctrl-R, but they start in regular-expression, non-incremental mode. You can also start a plain string search in non-incremental mode using the string-search and reverse-string-search commands. Some people like to bind these commands to Ctrl-S and Ctrl-R, respectively. Also see the `search-positions-at-start` variable.

Keep in mind that you can get from any type of search to any other type of search by typing the appropriate subcommands to a search. For example, if you meant to do a regex-search but instead typed Ctrl-S to do an incremental search, you could enter regex mode by typing Ctrl-T. Figure 4.1 summarizes the search subcommands.

Ctrl-S or Ctrl-R Switch to a new direction, or find the next occurrence in the same direction, or pull in the previous search string.

normal key Add that character to the search string.

⟨**Backspace**⟩ Remove the last character from the search string.

Ctrl-G Stop a running search, or (in incremental mode) delete characters until the search succeeds, or abort the search, returning to the starting point.

Ctrl-O Toggle incremental searching.

Ctrl-T Toggle regular expression searching.

Ctrl-W Toggle word searching. Matches must consist of complete words.

Ctrl-C Toggle case folding.

⟨**Enter**⟩ Exit the search.

Ctrl-Q Quote the following key, entering it into the search string even if it would normally run a command.

help key Show the list of search subcommands.

other keys If in incremental mode, exit the search, then execute the key normally. If not incremental mode, edit the search string.

Figure 4.1: The search subcommands work in all search and replace commands.

When you're at the last match of some text in a buffer, and tell incremental search to search again by pressing Ctrl-S, Epsilon displays "Failing" to indicate no more matches. If you press Ctrl-S once more, Epsilon will wrap to the beginning of the buffer and continue searching from there. It will display "Wrapped" to indicate it's done this. If you keep on search, eventually you'll pass your starting point again; then Epsilon will display "Overwrapped" to indicate that it's showing you a match you've already seen. A reverse search works similarly; Epsilon will wrap to the end of the buffer when you keep searching after a search has failed. (You can set the `search-wraps` variable to zero to disable wrapping.)

In some modes like Info mode, where a buffer displays a single part of some larger collection of text, pressing Ctrl-S at a failing search results in a continued search, instead of wrapping. Epsilon displays "Continued" to indicate (in the case of Info mode) that it's searching through other nodes.

The forward-search-again and reverse-search-again commands search forward and backward (respectively) for the last-searched-for search string, without prompting. The search-again command searches in the same direction as before for the same search string.

If you highlight a region before searching, Epsilon uses it as an initial search string if it's not very long. Set the `search-in-region` variable to make Epsilon instead restrict matches it finds to the highlighted region. Also see the `search-defaults-from` variable.

You can change the function of most keys in Epsilon by rebinding them (see page 126). But Epsilon doesn't implement the searching command keys listed above with the normal binding mechanism. The EEL code for searching refers directly to the keys Ctrl-C, Ctrl-W, Ctrl-T, Ctrl-O, Ctrl-Q, ⟨Enter⟩, and ⟨Esc⟩, so to change the function of these keys within searching you must modify the EEL code in the file search.e. Epsilon looks at your current bindings to determine which keys to use as the help key and backspace key. It looks at the `abort_key` variable to determine what to use as your abort key, instead of Ctrl-G. (See page 84.) Epsilon always recognizes Ctrl-S and Ctrl-R as direction keys, but you can set two variables `fwd-search-key` and `rev-search-key` to key codes. These will then act as "synonyms" to Ctrl-S and Ctrl-R, respectively.

When you select a searching command from the menu or tool bar (rather than via a command's keyboard binding), Epsilon for Windows runs the dialog-search or dialog-reverse-search command, to display a search dialog.

Most of the keys described above also work in dialog-based searching. However, dialog searching is never incremental, so Ctrl-O doesn't toggle incremental searching in a dialog. And Ctrl-Q doesn't quote the following character, because dialog searching doesn't support directly entering special characters.

To match special characters in dialog-based searching, you can enable regular expression searching, and then enter them using syntax like `<Tab>` or `<#13>`. See page 57. In replacement text, add a # first, as in `#<Newline>` or `#<#13>`. See page 64.

Summary:	Ctrl-S	incremental-search
	Ctrl-R	reverse-incremental-search
	Ctrl-Alt-S	regex-search
	Ctrl-Alt-R	reverse-regex-search
		string-search
		reverse-string-search
		search-again
		forward-search-again
		reverse-search-again
		dialog-search
		dialog-reverse-search

Searching Multiple Files

Epsilon provides a convenient grep command that lets you search a set of files. The command prompts you for a search string (all of the search options described above apply) and for a file pattern. By default, the grep interprets the search string as a regular expression (see page 55). To toggle regular expression mode, press Ctrl-T at any time while typing the search string. The command then scans the indicated files, puts a list of matching lines in the grep buffer, then displays the grep buffer in the current window. Each line indicates the file it came from.

With a numeric argument, this command searches through buffers instead of files. Instead of prompting for a file name pattern, Epsilon prompts for a buffer name pattern, and only operates on those buffers whose names match that pattern. Buffer name patterns use a simplified file name pattern syntax: * matches zero or more characters, ? matches any single character, and character classes like [a-z] may be used too.

When grep prompts for a file pattern, it shows you the last file pattern you searched inside square brackets. You can press ⟨Enter⟩ to conveniently search through the same files again. (See the `grep-default-directory` variable to control how Epsilon interprets this default pattern when the current directory has changed.)

By default file patterns you type are interpreted relative to the current buffer's file; see `grep-prompt-with-buffer-directory` to change this. To repeat a file pattern from before, press Alt-⟨Up⟩ or Ctrl-Alt-P. (See page 26 for details.) You can use extended file patterns to search in multiple directories; see page 108.

Epsilon skips over any file with an extension listed in `grep-ignore-file-extensions`; by default some binary file types are excluded. It also skips over files matched by the `grep-ignore-file-pattern` or `grep-ignore-file-basename` variables (the latter matched against just the base name of the file, not its path, the former matched against the entire file name). The `grep-ignore-file-types` variable makes grep skip over files that refer to devices, named pipes, or other sorts of special files. You can set the `use-grep-ignore-file-variables` variable to zero temporarily to have Epsilon ignore all these variables and search every matching file.

In a grep buffer, you can move around by using the normal movement commands. Most alphabetic keys run special grep commands. The 'N' and 'P' keys move to the next and previous matches.

You can easily go from the grep buffer to the corresponding locations in the original files. To do this, simply position point on the copy of the line, then press ⟨Space⟩, ⟨Enter⟩, or 'E'. The file appears in the current window, with point positioned at the beginning of the matching line. Typing '1' brings up the file in a window that occupies the entire screen. Typing '2' splits the window horizontally, then brings up the file in the lower window. Typing '5' splits the window vertically, then brings up the file. Typing the letter 'O' shows the file in the next window on the screen, without splitting windows any further. Typing 'Z' runs the zoom-window command, then brings up the file.

When Epsilon wants to search a particular file as a result of a grep command, it first scans the buffers to see if one of them contains the given file. If so, it uses that buffer. If the file doesn't appear in any buffer, Epsilon reads the file into a temporary buffer, does the search, then discards the buffer.

If you want Epsilon to always keep the files around in such cases, set the variable `grep-keeps-files` to a nonzero value. In that case, grep will simply use the find-file command to get any file it needs to search.

By default, each invocation of grep appends its results to the grep buffer. If you set the variable `grep-empties-buffer` to a nonzero value, grep will clear the grep buffer at the start of each invocation. Also see the `grep-show-absolute-path` variable to control the format of file names in the grep buffer.

You can move from match to match without returning to the grep buffer. The Ctrl-X Ctrl-N command moves directly to the next match. It does the same thing as switching to the grep buffer, moving down one line, then pressing ⟨Space⟩ to select that match. Similarly, Ctrl-X Ctrl-P backs up to the previous match.

Actually, Ctrl-X Ctrl-N runs the next-position command. After a grep command, this command simply calls next-match, which moves to the next match as described above. If you run a compiler in a subprocess, however, next-position calls next-error instead, to move to the next compiler error message. If you use the grep command again, or press ⟨Space⟩ in the grep buffer to select a match, or run next-match explicitly, then next-position will again call next-match to move to the next match.

Similarly, Ctrl-X Ctrl-P actually runs previous-position, which calls either previous-error or previous-match, depending upon whether you last ran a compiler or searched across files.

Summary:	Alt-F7	grep
	Ctrl-X Ctrl-N	next-position
	Ctrl-X Ctrl-P	previous-position
		next-match
		previous-match

4.2.4   Bookmarks

Epsilon's bookmark commands let you store the current editing position, so that you can easily return to it later. To drop a bookmark at point, use the Alt-/ key. For each bookmark, Epsilon remembers the buffer and the place within that buffer. Later, when you want to jump to that place, press Alt-J. Epsilon remembers the last 10 bookmarks that you set with Alt-/. To cycle through the last 10 bookmarks, you can press Alt-J and keep pressing it until you arrive at the desired bookmark.

You can set a named bookmark with the Ctrl-X / key. The command prompts you for a letter, then associates the current buffer and position with that letter. To jump to a named bookmark, use the Ctrl-X J key. It prompts you for the letter, then jumps to that bookmark.

Instead of a letter, you can specify a digit (0 to 9). In that case, the number refers to one of the temporary bookmarks that you set with the Alt-/ key. Zero refers to the last temporary bookmark, 1 to the one before that, and so on.

Whenever one of these commands asks you to specify a character for a bookmark, you can get a list by pressing '?'. Epsilon then pops up a list of the bookmarks you've defined, along with a copy of the line that contains the bookmark. You can simply move to one of the lines and press ⟨Enter⟩ to select that bookmark. In a list of bookmarks, press D to delete the highlighted bookmark.

The command list-bookmarks works like the Ctrl-X J key, but automatically pops up the list of bookmarks to choose from. If you like, you can bind it to Ctrl-X J to get that behavior.

Summary:	Alt-/	set-bookmark
	Alt-J	jump-to-last-bookmark
	Ctrl-X /	set-named-bookmark
	Ctrl-X J	jump-to-named-bookmark
		list-bookmarks

4.2.5   Tags

Epsilon provides a facility to remember which file defines a particular subroutine or procedure. This can come in handy if your program consists of several source files. Epsilon can remember this kind of information for you by using "tags". A tag instructs Epsilon to look for a particular function at a certain position in a certain file.

The goto-tag command on Ctrl-X ⟨Period⟩ prompts for the name of a function and jumps immediately to the definition of the routine. You can use completion (see page 24) while typing the tag name, or press '?' to select from a list of tags. (Epsilon also shows the defining file of each tag.)

If you don't give a name, goto-tag goes to the next tag with the same name as the last tag you gave it. If the same tag occurs several times (for example, if you tag several separate files that each define a `main()` function), use this to get to the other tag references, or press '?' after typing the tag name to select the correct file from a list. If you give goto-tag a numeric argument, it goes to the next tag without even asking for a name. When there are several instances of a single tag, you can also use Ctrl-⟨NumPlus⟩ and Ctrl-⟨NumMinus⟩ to move among them.

The pluck-tag command on Ctrl-X ⟨Comma⟩ first retrieves the routine name adjacent to or to the right of point, then jumps to that routine's definition.

If the file containing the definition appears in a window already, Epsilon will change to that window. Otherwise, Epsilon uses the find-file command to read the file into a buffer. Then Epsilon jumps to the definition, positioning its first line near the top of the window. You can set the window line to receive the

first line of the definition via the `show-tag-line` variable. It says how many lines down the definition should go.

Before Epsilon moves to the tag, it sets a temporary bookmark at your old position, just like the set-bookmark command on Alt-/. After goto-tag or pluck-tag, press Alt-J or Ctrl-⟨NumStar⟩ to move back to your previous position.

Normally, you have to tell Epsilon beforehand which files to look in. The tag-files command on Ctrl-X Alt-⟨Period⟩ prompts for a file name or file pattern such as *.c and makes a tag for each routine in the file. It knows how to recognize routines in C, C++, Java, Perl, Visual Basic, Python and EEL programs as well as assembly programs. (Using EEL, you can teach Epsilon to tag other languages too. See page 270.) If you tag a previously tagged file, the new tags replace all the old tags for that file. You can use extended file patterns to tag files in multiple directories; see page 108. When Epsilon can't find a tag, it tries retagging the current file before giving up; that means if your program is confined to one file, you don't have to tag it first. Set `tag-ask-before-retagging` nonzero if you want Epsilon to ask first.

In Perl, Visual Basic, and Python, Epsilon tags subroutine definitions. In C, C++, Java and EEL, tag-files normally tags subroutine and variable definitions, typedef definitions, structure and union member and tag definitions, enum constants, and `#define` constants. But it doesn't tag declarations (variables that use `extern`, function declarations without a body). With a numeric prefix argument, Epsilon includes these too. (Typically you'd do this for header files when you don't have source code for the function definitions—system files and library files, for instance.)

You can also set up tag-files to include declarations by default, by setting the `tag-declarations` variable. If zero (the default), tag-files only tags definitions. If one, Epsilon tags function declarations as well. If two, Epsilon tags variable declarations (which use the `extern` keyword). If three, Epsilon tags both types of declarations. Using a prefix argument with tag-files temporarily sets `tag-declarations` to three, so it tags everything it can. You can also set the `tag-which-items` variable to make tagging skip certain types of items, such as structure tag names or `#define` constants.

Set `tag-case-sensitive` nonzero if you want tagging to consider MAIN, Main and main to be distinct tags. By default, typing "main" will find any of these.

Epsilon can maintain separate groups of tags, each in a separate file. The select-tag-file command on Ctrl-X Alt-⟨Comma⟩ prompts for the name of a tag file, and uses that file for tag definitions.

When Epsilon needs to find a tag file, it searches for a file in the current directory, then in its parent directory, then in that directory's parent, and so forth, until it reaches the root directory or finds a file "default.tag". If Epsilon finds no file with that name, it creates a new tag file in the current directory. To force Epsilon to create a new tag file in the current directory, even if a tag file exists in a parent directory, use the select-tag-file command.

You can set the variable `initial-tag-file` to a relative pathname like "myfile.tag", if you want Epsilon to search for that file, or you can set it to an absolute pathname if you want Epsilon to use the same tag file no matter which directory you use.

The tag system can also use .BSC files from Microsoft Visual C++ 4.1 and later. Epsilon requires a Microsoft Browser Toolkit DLL to do this. We have not received permission from Microsoft to redistribute this DLL, but you can download it from their web site. See http://www.lugaru.com/links.html#bsc for details.

If you use Visual 4.1 or 4.2, download the archive BSCKIT41.EXE and extract the file bsc41.dll. If you use Visual C 5.0, download the archive BSCKIT50.EXE and extract the file msbsc50.dll. If you use Visual C 6.0, download the archive BSCKIT60.EXE and extract the file msbsc60.dll. If you use Visual C .NET, download the archive BSCKIT70.EXE and extract the file msbsc70.dll. With any of these DLL files, rename it to bsc.dll and place it in the directory containing Epsilon's executable (for example c:\Program Files\Eps12\Bin). Then use the select-tag-file command on Ctrl-X Alt-⟨Comma⟩ to select your .BSC file.

When Epsilon uses a .BSC file, the commands tag-files, retag-files, clear-tags, sort-tags, and the variables `tag-case-sensitive`, `tag-relative`, `want-sorted-tags`, and `tag-by-text` do not apply. See Microsoft compiler documentation for information on generating .BSC and .SBR files.

The retag-files command makes Epsilon rescan all the files represented in the current tag file and generate a new set of tags for each, replacing any prior tags. The clear-tags command makes Epsilon forget about all the tags in the current tag file. The untag-files command displays a list of all files mentioned in the current tag file; you can edit the list by deleting any file names that shouldn't be included, and when you press Ctrl-X Ctrl-Z, Epsilon will forget all tags that refer to the file names you deleted.

When Epsilon records a tag, it stores the character position and the text of the line at the tag position. If the tag doesn't appear at the remembered character offset, Epsilon searches for the defining line. And if that doesn't work (perhaps because its defining line has changed) Epsilon retags the file and tries again. This means that once you tag a file, it should rarely prove necessary to retag it, even if you edit the file. To save space in the tag file, you can have Epsilon record only the character offset, by setting the variable `tag-by-text` to zero. Because this makes Epsilon's tagging mechanism faster, it's a good idea to turn off `tag-by-text` before tagging any very large set of files that rarely changes.

By default, Epsilon sorts the tag list whenever it needs to display a list of tag names for you to choose from. Although Epsilon tries to minimize the time taken to sort this list, you may find it objectionable if you have many tags. Instead, you can set the `want-sorted-tags` variable to 0, and sort the tags manually, whenever you want, using the sort-tags command.

Epsilon normally stores file names in its tag file in relative format, when possible. This means if you rename or copy a directory that contains some source files and a tag file for them, the tag file will still work fine. If you set the variable `tag-relative` to 0, Epsilon will record each file name with an absolute pathname instead.

Summary:	Ctrl-X ⟨Period⟩	goto-tag
	Ctrl-X ⟨Comma⟩	pluck-tag
	Ctrl-X Alt-⟨Period⟩	tag-files
	Ctrl-X Alt-⟨Comma⟩	select-tag-file
	Ctrl-⟨NumPlus⟩	next-tag
	Ctrl-⟨NumMinus⟩	previous-tag
		retag-files
		clear-tags
		untag-files
		sort-tags

4.2.6 Comparing

The compare-windows command on Ctrl-F2 finds differences between the contents of the current buffer and that displayed in the next window on the screen. If called while in the last window, it compares that window with the first window. The comparison begins at point in each window. Epsilon finds the first difference between the buffers and moves the point to just before the differing characters, or to the ends of the buffers if it finds no difference. It then displays a message in the echo area reporting whether or not it found a difference.

If you invoke compare-windows again immediately after it has found a difference, the command will try to resynchronize the windows by moving forward in each window until it finds a match of at least `resynch-match-chars` characters. It doesn't necessarily move each window by the same amount, but

instead finds a match that minimizes the movement in the window that it moves the most. It then reports the number of characters in each window it skipped past.

Normally compare-windows treats one run of space and tab characters the same as any other run, so it skips over differences in horizontal whitespace. You can set the `compare-windows-ignores-space` variable to change this.

The diff command works like compare-windows, but it will compare and resynchronize over and over from the beginning to the end of each buffer, producing a report that lists all differences between the two buffers. It operates line-by-line rather than character-by-character.

When resynchronizing, diff believes it has found another match when `diff-match-lines` lines in a row match, and gives up if it cannot find a match within `diff-mismatch-lines` lines. By default, diff resynchronizes when it encounters three lines in a row that match. Under Windows and Unix, normally Epsilon uses a smarter algorithm that's better at finding a minimum set of differences. With this algorithm, `diff-mismatch-lines` isn't used. But because this algorithm becomes very slow when buffers are large, it's only used when at least one of the buffers contains fewer than `diff-precise-limit` bytes (by default 500 KB).

The diff command reports each difference with a summary line and then the text of the differing lines. The summary line consists of two line number ranges with a letter between them indicating the type of change: 'a' indicates lines to add to the first buffer to match the second, 'd' indicates lines to delete, and 'c' indicates lines to change. For example, a summary line in the diff listing of "20,30c23,29" means to remove lines 20 through 30 from the first buffer and replace them with a copy of lines 23 through 29 from the second buffer. "11a12" means that adding line 12 from the second buffer right after line 11 in the first buffer would make them identical. "11,13d10" means that deleting lines 11, 12 and 13 from the first buffer (which would appear just after line 10 in the second) would make them identical.

After each summary line, diff puts the lines to which the summary refers. The diff command prefixes lines to delete from the first buffer by "<" and lines to add by ">".

The visual-diff command is like diff but uses colors to show differences. It constructs a new buffer that contains all the lines of the two buffers. Lines from the first buffer that don't appear in the second are displayed with a red background. Lines in the second buffer that don't appear in the first have a yellow background. Lines that are the same in both buffers are colored normally.

This command also does character-by-character highlighting for each group of changed lines. Instead of simply indicating that one group of lines was replaced by another, it shows which portions of the lines changed and which did not, by omitting the red or yellow background from those characters. You can set the variables `diff-match-characters` and `diff-match-characters-limit` to alter or turn off this behavior.

In a visual-diff buffer, the keys Alt-⟨Down⟩ and Alt-] move to the start of the next changed or common section. The keys Alt-⟨Up⟩ and Alt-[move to the previous change or common section.

The merge-diff command is another variation on diff that's useful with buffers in C mode. It marks differences by surrounding them with #ifdef preprocessor lines, first prompting for the #ifdef variable name to use. The resulting buffer receives the mode and settings of the first of the original buffers. The marking is mechanical, and doesn't parse the text being marked off, so it may produce invalid code. For example, if an #if statement differs between the two buffers, the result will contain improperly nested #if statements like this:

```
#ifndef DIFFVAR
     #if DOSVERSION
#else // DIFFVAR
     #if MSDOSVERSION
#endif // DIFFVAR
```

Therefore, you should examine the output of merge-diff before trying to compile it.

Like compare-windows and diff, the compare-sorted-windows command compares the contents of the current buffer with that displayed in the next window on the screen. Use it when you have (for example) two lists of variable names, and you want to find out which variables appear on only one or the other list, and which appear on both. This command assumes that you sorted both the buffers. It copies all lines appearing in both buffers to a buffer named "inboth". It copies all lines that appear only in the first buffer to a buffer named "only1", and lines that appear only in the second to a buffer named "only2".

The uniq command goes through the current buffer and looks for adjacent identical lines, deleting the duplicate copies of each repeated line and leaving just one. It doesn't modify any lines that only occur once. This command behaves the same as the Unix command of the same name.

The keep-unique-lines command deletes all copies of any duplicated lines. This command acts like the Unix command "uniq -u".

The keep-duplicate-lines command deletes all lines that only occur once, and leaves one copy of each duplicated line. This command acts like the Unix command "uniq -d".

The following table shows how sample text would be modified by each of the above commands.

Sample text	Uniq	Keep-duplicate-lines	Keep-unique-lines
dog	dog	dog	cat
dog	cat	horse	rabbit
cat	horse		dog
horse	rabbit		
horse	dog		
horse			
rabbit			
dog			

Summary: Ctrl-F2, Ctrl-X C   compare-windows

compare-sorted-windows

diff

visual-diff

visual-diff-mode

merge-diff

uniq

keep-unique-lines

keep-duplicate-lines

Visual Diff only: Alt-], Alt-⟨Down⟩   next-difference

Visual Diff only: Alt-[, Alt-⟨Up⟩   previous-difference

4.3 Changing Text

4.3.1 Inserting and Deleting

When you type most alphabetic or numeric keys, they appear in the buffer before point. Typing one of these keys runs the command normal-character, which simply inserts the character that invoked it into the buffer.

When you type a character bound to the normal-character command, Epsilon inserts the character before point, so that the cursor moves forward as you type characters. Epsilon can also overwrite as you type. The overwrite-mode command, bound to the ⟨Ins⟩ key, toggles overwriting for the current buffer. If you give it a nonzero numeric argument (for example, by typing Ctrl-U before invoking the command, see page 22), it doesn't toggle overwriting, but turns it on. Similarly, a numeric argument of zero always turns off overwriting. Overwriting will occur for all characters except newline, and overwriting never occurs at the end of a line. In these cases the usual insertion will happen. The buffer-specific variable `over-mode` controls overwriting.

The Ctrl-Q key inserts special characters, such as control characters, into the current buffer. It waits for you to type a character, then inserts it. This command ignores keys that don't represent characters, such as ⟨Home⟩ or F3. If you "quote" an Alt key in this way, Epsilon inserts the corresponding character with its high bit on. You can use this command for inserting characters like Ctrl-Z that would normally execute a command when typed.

Sometimes you may want to insert a character whose numeric ASCII value you know, but you may not know which keystroke that character corresponds to. Epsilon provides an insert-ascii command on Alt-# for this purpose. It prompts you for a numeric value, then inserts the character with that value into the buffer. By default, the command interprets the value in base 10. You can specify a hexadecimal value by prefixing the characters "0x" to the number, or an octal value by prefixing the character "0o" to the number, or a binary value by prefixing "0b". For example, the numbers "87", "0x57", "0o127", and "0b1010111" all refer to the same number, and they all would insert a "W" character if given to the insert-ascii command. You can also use the name of a Unicode character inside angle brackets, like "`<square root>`". You can use completion on character names like this.

In most environments you can type graphics characters by holding down the Alt key and typing the character's value on the numeric keypad, but see the `alt-numpad-keys` variable. In some environments, Epsilon will automatically quote the character so that it's inserted in the buffer and not interpreted as a command. (You may need to type a Ctrl-Q first to quote the character in other environments.)

The Ctrl-O command inserts a newline after point (or, to put it another way, inserts a newline before point as usual, then backs up over it). Use this command to break a line when you want to insert new text in the middle, or to "open" up some space after point.

The ⟨Backspace⟩ key deletes the character before point, and the ⟨Del⟩ key deletes the character after point. In other words, ⟨Backspace⟩ deletes backwards, and ⟨Del⟩ deletes forwards. These commands usually do not save deleted characters in the kill ring (see the next section).

If you prefix these commands with a numeric argument of n, they will delete n characters instead of one. In that case, you can retrieve the deleted text from the kill ring with the Ctrl-Y key (see the next section).

If ⟨Backspace⟩ or ⟨Del⟩ follows one of the kill commands, the deleted character becomes part of the text removed by the kill command. See the following section for information on the kill commands.

The buffer-specific variable `delete-hacking-tabs` makes ⟨Backspace⟩ operate differently when deleting tabs or spaces. If 1, when ⟨Backspace⟩ deletes a tab, it first turns the tab into the number of spaces necessary to keep the cursor in the same column, then deletes one of the spaces. If 2, when ⟨Backspace⟩ deletes a space, it deletes additional spaces and tabs until it reaches the previous tab column. The first setting makes ⟨Backspace⟩ treat tabs more like spaces; the second makes it treats spaces more like tabs.

The key Alt-\ deletes spaces and tabs surrounding point.

The Ctrl-X Ctrl-O command deletes empty lines adjacent to point, or lines that contain only spaces and tabs, turning two or more such blank lines into a single blank line. Ctrl-X Ctrl-O deletes a lone blank line. If you prefix a numeric argument of n, exactly n blank lines appear regardless of the number of blank lines present originally.

Summary:	Ctrl-Q	quoted-insert
	Alt-#	insert-ascii
	Ctrl-O	open-line
	Ctrl-H, ⟨Backspace⟩	backward-delete-character
	Ctrl-D, ⟨Del⟩	delete-character
	Alt-\	delete-horizontal-space
	Ctrl-X Ctrl-O	delete-blank-lines
	"normal keys"	normal-character
	⟨Ins⟩	overwrite-mode

4.3.2 The Region, the Mark, and Killing

Epsilon has many commands to erase characters from a buffer. Some of these commands save the erased characters away in a special group of buffers called *kill buffers*, and some do not.

In Epsilon's terminology, to *kill* means to delete text and save it away in a kill buffer, and to *delete* means simply to remove the text and not save it away. Any consecutive sequence of killing commands will produce a single block of saved text. The Ctrl-Y command then yanks back the entire block of text, inserting it before point. (Even when Epsilon deletes text and doesn't save it, you can usually use the undo command to recover the text. See page 83.)

The Ctrl-K command kills to the end of the line, but does not remove the line separator. At the end of a line, though, it kills just the line separator. Thus, use two Ctrl-K's to completely remove a nonempty line. Give this command a numeric argument of n to kill exactly n lines, including the line separators. If you give the Ctrl-K command a negative numeric argument, $-n$, the command kills from the beginning of the previous nth line to point.

The kill-current-line command is an alternative to Ctrl-K. It kills the entire line in one step, including the line separator. The kill-to-end-of-line command kills the rest of the line. If point is at the end of the line, it does nothing. In Brief mode Epsilon uses these two commands in place of the kill-line command that's normally bound to Ctrl-K.

The commands to delete single characters will also save the characters if you give them a numeric argument (to delete that number of characters) or if they follow a command which itself kills text.

Several Epsilon commands operate on a *region* of text. To specify a region, move to either end of the region and press the Ctrl-@ key or the Ctrl-⟨Space⟩ key. This sets the *mark* to the current value of point. Then move point to the other end of the region. The text between the mark and point specifies the region.

When you set the mark with Ctrl-@, Epsilon turns on highlighting for the region. As you move point away from the mark, the region appears in a highlighted color. This allows you to see exactly what text a region-sensitive command would operate upon. To turn the highlighting off, type Ctrl-X Ctrl-H. The Ctrl-X Ctrl-H command toggles highlighting for the region. If you prefix a nonzero numeric argument, it turns highlighting on; a numeric argument of zero turns highlighting off.

You can also check the ends of the region with the Ctrl-X Ctrl-X command. This switches point and mark, to let you see the other end of the region. Most commands do not care whether point (or mark) refers to the beginning or the end of the region.

The mark-whole-buffer command on Ctrl-X H provides a quick way to set point and mark around the entire buffer.

Another way to select text is to hold down the Shift key and move around using the arrow keys, or the keys ⟨Home⟩, ⟨End⟩, ⟨PageUp⟩, or ⟨PageDown⟩. Epsilon will select the text you move through. The `shift-selects` variable controls this feature.

The Ctrl-W command kills the region, saving it in a kill buffer. The Ctrl-Y command then yanks back the text you've just killed, whether by the Ctrl-W command or any other command that kills text. It sets the region around the yanked text, so you can kill it again with a Ctrl-W, perhaps after adjusting the region at either end. The Alt-W command works like Ctrl-W, except that it does not remove any text from the buffer; it simply copies the text between point and mark to a kill buffer.

Each time you issue a sequence of killing commands, Epsilon saves the entire block of deleted text as a unit in one of its kill buffers. The Ctrl-Y command yanks back the last of these blocks. To access the other blocks of killed text, use the Alt-Y command. It follows a Ctrl-Y or Alt-Y command, and replaces the retrieved text with an earlier block of killed text. Each time you press Alt-Y, Epsilon substitutes a block from another kill buffer, cycling from most recent back through the oldest, and then around to the most recent again.

In normal use, you go to the place you want to insert the text and issue the Ctrl-Y command. If this doesn't provide the right text, give the Alt-Y command repeatedly until you see the text you want. If the text you want does not appear in any of the killed blocks, you can get rid of the block with Ctrl-W, since both Ctrl-Y and Alt-Y always place point and mark around the retrieved block.

By default, Epsilon provides ten kill buffers. You can set the variable `kill-buffers` if you want a different number of kill buffers. Setting this variable to a new value makes Epsilon throw away the contents of all the kill buffers the next time you execute a command that uses kill buffers.

The Alt-Y command doesn't do anything if the region changed since the last Ctrl-Y or Alt-Y, so you can't lose text with a misplaced Alt-Y. Neither of these commands changes the kill buffers themselves. The Alt-Y command uses the undo facility, so if you've disabled undo, it won't work.

Epsilon can automatically reindent yanked text. By default it does this in C mode buffers. See page 67 for details. If you invoke Ctrl-Y or Alt-Y with a negative numeric prefix argument, by typing Alt-⟨Minus⟩ Ctrl-Y for example, the command won't reindent the yanked text, and will insert one copy. (Providing a positive numeric prefix argument makes Epsilon yank that many copies of the text. See page 124.)

Each time you issue a sequence of killing commands, all the killed text goes into one kill buffer. When a killing command follows a non-killing command, the text goes into a new kill buffer (assuming you haven't set up Epsilon to have only one kill buffer). You may sometimes want to append a new kill to the current kill buffer, rather than using the next kill buffer. That would let you yank all the text back at once. The Ctrl-Alt-W command makes an immediately following kill command append to a kill buffer instead of moving to a new one.

The Ctrl-Y command can come in handy when entering text for another command. For example, suppose the current buffer contains a line with "report.txt" on it, and you now want to read in the file with that name. Simply kill the line with Ctrl-K and yank it back (so as not to change the buffer) then give the Ctrl-X Ctrl-F command (see page 95) to read in a file. When prompted for the file name, press Ctrl-Y and the text "report.txt" appears as if you typed it yourself.

Pressing a self-inserting key like 'j' while text is highlighted normally deletes the highlighted selection, replacing it with the key. Pressing ⟨Backspace⟩ simply deletes the text. You can disable this behavior by setting the variable `typing-deletes-highlight` to zero. If you turn off this feature, you may also wish to set the variable `insert-default-response` to zero. At many prompts Epsilon will insert a highlighted default response before you start typing, if this variable is nonzero. You may also wish to set `typing-hides-highlight` if you've disabled `typing-deletes-highlight`, so pressing a self-inserting key turns off highlighting but doesn't delete anything.

In addition to the above commands which put the text into temporary kill buffers, Epsilon provides commands to make more permanent copies of text. The Ctrl-X X key copies the text in the region between point and mark to a permanent buffer. The command prompts you for a letter (or number), then associates the text with that letter. Thereafter, you can retrieve the text using the Ctrl-X Y key. That command asks you for the letter, then inserts the corresponding text before point.

Summary:	Ctrl-@, Alt-@	set-mark
	Ctrl-X Ctrl-H	highlight-region
	Ctrl-X Ctrl-X	exchange-point-and-mark
	Ctrl-K	kill-line
	Ctrl-W	kill-region
	Alt-W	copy-region
	Ctrl-Y	yank
	Alt-Y	yank-pop
	Ctrl-Alt-W	append-next-kill
	Ctrl-X X	copy-to-scratch
	Ctrl-X Y	insert-scratch
	Ctrl-X H	mark-whole-buffer
		kill-current-line
		kill-to-end-of-line

4.3.3 Clipboard Access

In Windows, Epsilon's killing commands interact with the Windows clipboard. Similarly, Epsilon for Unix interacts with the X clipboard when running as an X program. You can kill text in Epsilon and paste it into another application, or copy text from an application and bring it into Epsilon with the yank command.

All commands that put text on the kill ring will also try to copy the text to the clipboard, if the variable `clipboard-access` is non-zero. You can copy the current region to the clipboard without putting it on the kill ring using the command copy-to-clipboard.

The yank command copies new text from the clipboard to the top of the kill ring. It does this only when the clipboard's contents have changed since the last time Epsilon accessed it, the clipboard contains text, and `clipboard-access` is non-zero. Epsilon looks at the size of the clipboard to determine if the text on it is new, so it may not always notice new text. You can force Epsilon to retrieve text from the clipboard by using the insert-clipboard command, which inserts the text on the clipboard at point in the current buffer.

If you prefer to have Epsilon ignore the clipboard except when you explicitly tell it otherwise, set `clipboard-access` to zero. You can still use the commands copy-to-clipboard and insert-clipboard to work with the clipboard. Unlike the transparent clipboard support provided by `clipboard-access`, these commands will report any errors that occur while trying to access the clipboard. If transparent clipboard support cannot access the clipboard for any reason, it won't report an error, but will simply ignore the clipboard. Epsilon also disables transparent clipboard support when running a keyboard macro, unless `clipboard-access` is 2.

By default, when the Win32 Console version of Epsilon puts characters on the clipboard, it lets Windows translate the characters from the OEM character set to Windows ANSI, so that national characters display correctly. Epsilon for Windows uses Windows ANSI like other Windows programs, so no translation is needed. See the description of the `clipboard-format` variable to change this.

Summary:		copy-to-clipboard
		insert-clipboard

4.3.4 Rectangle Commands

Epsilon regions actually come in four distinct types. Each type has a corresponding Epsilon command that begins defining a region of that type.

Region Type	Command
Normal	mark-normal-region
Line	mark-line-region
Inclusive	mark-inclusive-region
Rectangular	mark-rectangle

The commands are otherwise very similar. Each command starts defining a region of the specified type, setting the mark equal to point and turning on highlighting. If Epsilon is already highlighting a region of a different type, these commands change the type. If Epsilon is already highlighting a region of the same type, these commands start defining a new region by setting mark to point again. (You can set the variable `mark-unhighlights` to make the commands turn off the highlighting and leave the mark alone in this case.)

The mark-normal-region command defines the same kind of region as the set-mark command described in section 4.3.2. (The commands differ in that set-mark always begins defining a new region, even if another type of region is highlighted on the screen. The mark-normal-region command converts the old region, as described above.)

A line region always contains entire lines of text. It consists of the line containing point, the line containing mark, and all lines between the two.

An inclusive region is very similar to a normal region, but an inclusive region contains one additional character at the end of the region. A normal region contains all characters between point and mark, if you think of point and mark as being positioned between characters. But if you think of point and mark as character positions, then an inclusive region contains the character at point, the character at the mark, and all characters between the two. An inclusive region always contains at least one character (unless point and mark are both at the end of the buffer).

A rectangular region consists of all columns between those of point and mark, on all lines in the buffer between those of point and mark. The mark-rectangle command on Ctrl-X # begins defining a rectangular region. In a rectangular region, point can specify any of the four corners of this rectangle.

Some commands operate differently when the current region is rectangular. Killing a rectangular region by pressing the Ctrl-W key runs the command kill-rectangle. It saves the current rectangle in a kill buffer, and replaces the rectangle with spaces, so as not to shift any text that appears to the right of the rectangle. (But see the `kill-rectangle-removes` variable.)

The Alt-W key runs the command copy-rectangle. It also saves the current rectangle, but doesn't modify the buffer. (Actually, it may insert spaces at the ends of lines, or convert tabs to spaces, if that's necessary to reach the starting or ending column on one of the lines in the region. But the buffer won't look any different as a result of these changes. Most rectangle commands do this.)

The Ctrl-Alt-W key runs the command delete-rectangle. It removes the current rectangle, shifting any text after it to the left. It doesn't save the rectangle.

When you use the Ctrl-Y key to yank a kill buffer that contains a rectangle, Epsilon inserts the last killed rectangle into the buffer at the current column, on the current and successive lines. It shifts existing text to the right. If you've enabled overwrite mode, however, the rectangle replaces any existing text in those columns. See the `yank-rectangle-to-corner` variable to set how Epsilon positions point and mark around the yanked rectangle. You can use the Alt-Y key to cycle through previous kills as usual.

When yanking line regions, the `yank-line-retains-position` variable serves a similar purpose, influencing where Epsilon positions the cursor.

The width of a tab character depends upon the column it occurs in. For this reason, if you use the rectangle commands to kill or copy text containing tabs, and you move the tabs to a different column, text after the tabs may shift columns. (For example, a tab at column 0 occupies 8 columns, but a tab at column 6 occupies only 2 columns.) You can avoid this problem by using spaces instead of tabs with the rectangle commands.

The buffer-specific variable `indent-with-tabs` controls whether Epsilon does indenting with tabs or only with spaces. Set it to 0 to make Epsilon always use spaces. This variable affects only future indenting you may do; it doesn't change your file. To replace the tabs in your file, use the untabify-buffer command.

Summary:	Ctrl-X #	mark-rectangle
	Ctrl-W	kill-rectangle
	Alt-W	copy-rectangle
	Ctrl-Alt-W	delete-rectangle
		mark-line-region
		mark-inclusive-region

4.3.5 Capitalization

Epsilon has commands that allow you to change the case of words. Each travels forward, looking for the end of a word, and changes the case of the letters it travels past. Thus, if you give these commands while inside a word, only the rest of the word potentially changes case.

The Alt-L key, lowercase-word, turns all the characters it passes to lower case. The Alt-U key, uppercase-word, turns them all to upper case. The Alt-C key, capitalize-word, capitalizes a word by making the first letter it travels past upper case, and all the rest lower case. All these commands position point after the word operated upon.

For example, the Alt-L command would turn "wOrd" into "word". The Alt-U command would turn it into "WORD", and the Alt-C command would turn it into "Word".

These commands operate on the highlighted region, if there is one. If there is no highlighted region, the commands operate on the next word and move past it, as described above. The commands work on both conventional and rectangular regions.

Summary:	Alt-C	capitalize-word
	Alt-L	lowercase-word
	Alt-U	uppercase-word

4.3.6 Replacing

The key Alt-& runs the command replace-string, and allows you to change all occurrences of a string in the rest of your document to another string. Epsilon prompts for the string to replace, and what to replace it with. Terminate the strings with ⟨Enter⟩. After you enter both strings, Epsilon replaces all occurrences of the first string after point with instances of the second string (but respecting any narrowing restriction; see page 143).

When entering the string to search for, you can use any of the searching subcommands described on page 39: Ctrl-C toggles case-folding, Ctrl-W toggles word searching, and Ctrl-T toggles interpreting the string as a regular expression.

To enter special characters in either the search or replace strings, use Ctrl-Q before each. Type Ctrl-Q Ctrl-C to include a Ctrl-C character. Type Ctrl-Q Ctrl-J to include a ⟨Newline⟩ character in a search string or replacement text. Press Alt-g when entering the replacement string to copy the search string.

The key Alt-R runs the command query-replace, which works like replace-string. Instead of replacing everything automatically, however, the command positions point after each occurrence of the old string and waits for you to press a key. You may choose whether to replace this occurrence or not:

y or Y or ⟨Space⟩ Replace it, go on to next occurrence.

n or N or ⟨Backspace⟩ Don't replace it, go on to next occurrence.

! Replace all remaining occurrences. The replace-string command works like the query-replace command followed by pressing '!' when it shows you the first match.

⟨Esc⟩ Exit and leave point at the match in the buffer.

ˆ Back up to the previous match.

⟨Period⟩ Replace this occurrence and then exit.

⟨Comma⟩ Replace and wait for another command option without going on to the next match.

Ctrl-R Enter a recursive edit. Point and mark go around the match. You may edit arbitrarily. When you exit the recursive edit with Ctrl-X Ctrl-Z, Epsilon restores the old mark, and the query-replace continues from the current location.

Ctrl-G Exit and restore point to its original location.

Ctrl-T Toggle regular expression searching. See the next section for an explanation of regular expressions.

Ctrl-W Toggle word searching.

Ctrl-C Toggle case folding.

? or help key Provide help, including a list of these options.

anything else Exit the replacement, staying at the current location, and execute this key as a command.

The command regex-replace operates like query-replace, but starts up in regular expression mode. See page 64.

The command reverse-replace operates like query-replace, but moves backwards. You can also trigger a reverse replacement by pressing Ctrl-R while entering the search text for any of the replacing commands.

If you invoke any of the replacing commands above with a numeric argument, Epsilon will use word searching.

If you highlight a region before replacing, Epsilon uses it as an initial search string if it's not very long. Set the `replace-in-region` variable to make Epsilon instead restrict its replacements to the highlighted region. Also see the `search-defaults-from` variable.

Replace commands preserve case. Epsilon examines the case of each match. If a match is entirely upper case, or all words are capitalized, Epsilon makes the replacement text entirely upper case or capitalized, as appropriate. Epsilon only does this when searching is case-insensitive, and neither the search string nor the replace string contain upper case letters. For example, if you search for the regular expression `welcome|hello` and replace it with `greetings`, Epsilon replaces HELLO with GREETINGS and Welcome with Greetings. See the `replace-by-case` variable to alter the rules Epsilon uses. With a regular expression replace, you can force parts of the replacement to a particular case; see page 64.

The file-query-replace command on Shift-F7 replaces text in multiple files. It prompts for the search text, replacement text, and a file name which may contain wildcards. You can use extended file patterns to replace in files from multiple directories; see page 108. Epsilon skips over any file with an extension listed in `grep-ignore-file-extensions` or meeting other criteria, just like the grep command. See page 42 for details. To search without replacing, see the grep command on page 41.

With a numeric argument, this command searches through buffers instead of files. Instead of prompting for a file name pattern, Epsilon prompts for a buffer name pattern, and only operates on those buffers whose names match that pattern. Buffer name patterns use a simplified file name pattern syntax: ∗ matches zero or more characters, ? matches any single character, and character classes like [a-z] may be used too.

The command delete-matching-lines prompts for a regular expression pattern. It then deletes all lines after point in the current buffer that contain the pattern. The similar command keep-matching-lines deletes all lines *except* those that contain the pattern. As with any searching command, you can press Ctrl-T, Ctrl-W, or Ctrl-C while typing the pattern to toggle regular expression mode, word mode, or case folding (respectively).

When you select a replacing command from the menu or tool bar (rather than via a command's keyboard binding), Epsilon for Windows runs the dialog-replace or dialog-regex-replace command, to display a replace dialog. Controls on the dialog replace many of the keys described above.

Summary:	Alt-&	replace-string
	Alt-R, Alt-%	query-replace
	Shift-F7	file-query-replace
	Alt-*	regex-replace
		reverse-replace
		delete-matching-lines
		keep-matching-lines

4.3.7 Regular Expressions

Most of Epsilon's searching commands, described on page 38, take a simple string to search for. Epsilon provides a more powerful regular expression search facility, and a regular expression replace facility.

Instead of a simple search string, you provide a pattern, which describes a set of strings. Epsilon searches the buffer for an occurrence of one of the strings contained in the set. You can think of the pattern as generating a (possibly infinite) set of strings, and the regex search commands as looking in the buffer for the first occurrence of one of those strings.

The following characters have special meaning in a regex search: vertical bar, parentheses, plus, star, question mark, square brackets, period, dollar, percent sign, left angle bracket ('<'), and caret ('^'). To match them literally, they must be quoted; see page 57. See the following sections for syntax details and additional examples.

PLAIN PATTERNS.

In a regular expression, a string that does not contain any of the above characters denotes the set that contains precisely that one string. For example, the regular expression abc denotes the set that contains, as its only member, the string 'abc'. If you search for this regular expression, Epsilon will search for the string 'abc', just as in a normal search.

ALTERNATION.

To include more than one string in the set, you can use the vertical bar character. For example, the regular expression abc|xyz denotes the set that contains the strings 'abc' and 'xyz'. If you search for that

`abc|def`	Finds either `abc` or `def`.
`(abc)`	Finds `abc`.
`abc+`	Finds `abc` or `abcc` or `abccc` or . . . .
`abc*`	Finds `ab` or `abc` or `abcc` or `abccc` or . . . .
`abc?`	Finds `ab` or `abc`.
`[abcx-z]`	Finds any single character of a, b, c, x, y, or z.
`[^abcx-z]`	Finds any single character except a, b, c, x, y, or z.
`.`	Finds any single character except ⟨Newline⟩.
`abc$`	Finds `abc` that occurs at the end of a line.
`^abc`	Finds `abc` that occurs at the beginning of a line.
`%^abc`	Finds a literal `^abc`.
`<Tab>`	Finds a ⟨Tab⟩ character.
`<#123>`	Finds the character with ASCII code 123.
`<p:cyrillic>`	Finds any character with that Unicode property.
`<alpha|1-5&!x-z>`	Finds any alpha character except x, y or z or digit 1–5.
`<^c:*comment>printf`	Finds uses of printf that aren't commented out.
`<h:0d 0a 45>`	Finds char sequence with those hexadecimal codes.

Figure 4.2: Summary of regular expression characters.

pattern, Epsilon will find the first occurrence of either 'abc' or 'xyz'. The alternation operator (`|`) always applies as widely as possible, limited only by grouping parentheses.

GROUPING.

You can enclose any regular expression in parentheses, and the resulting expression refers to the same set. So searching for `(abc|xyz)` has the same effect as searching for `abc|xyz`, which works as in the previous paragraph. You would use parentheses for grouping purposes in conjunction with some of the operators described below.

CONCATENATION.

You can concatenate two regular expressions to form a new regular expression. Suppose the regular expressions p and q denote sets P and Q, respectively. Then the regular expression pq denotes the set of strings that you can make by concatenating, to members of P, strings from the set Q. For example, suppose you concatenate the regular expressions `(abc|xyz)` and `(def|ghi)` to yield `(abc|xyz)(def|ghi)`. From the previous paragraph, we know that `(abc|xyz)` denotes the set that contains 'abc' and 'xyz'; the expression `(def|ghi)` denotes the set that contains 'def' and 'ghi'. Applying the rule, we see that `(abc|xyz)(def|ghi)` denotes the set that contains the following four strings: 'abcdef', 'abcghi', 'xyzdef', 'xyzghi'.

CLOSURE.

Clearly, any regular expression must have finite length; otherwise you couldn't type it in. But because of the closure operators, the set to which the regular expression refers may contain an infinite number of strings. If you append plus to a parenthesized regular expression, the resulting expression denotes the set of one or more repetitions of that string. For example, the regular expression `(ab)+` refers to the set that contains 'ab', 'abab', 'ababab', 'abababab', and so on. Star works similarly, except it denotes the set of zero or more repetitions of the indicated string.

S<small>MALL</small> O<small>PTIONALITY</small>.

You can specify the question operator in the same place you might put a star or a plus. If you append a question mark to a parenthesized regular expression, the resulting expression denotes the set that contains that string, and the empty string. You would typically use the question operator to specify an optional subpart of the search string.

You can also use the plus, star, and question-mark operators with subexpressions, and with non-parenthesized things. These operators always apply to the smallest possible substring to their left. For example, the regular expression `abc+` refers to the set that contains 'abc', 'abcc', 'abccc', 'abcccc', and so on. The expression `a(bc)*d` refers to the set that contains 'ad', 'abcd', 'abcbcd', 'abcbcbcd', and so on. The expression `a(b?c)*d` denotes the set that contains all strings that start with 'a' and end with 'd', with the inside consisting of any number of the letter 'c', each optionally preceded by 'b'. The set includes such strings as 'ad', 'acd', 'abcd', 'abccccbcd'.

Entering Special Characters

`<Comma>`	,	`<Nul>`	^@	`<Period>`	.
`<Space>`		`<Star>`	*	`<Plus>`	+
`<Enter>`	^M	`<Percent>`	%	`<Vbar>`	|
`<Return>`	^M	`<Lparen>`	(	`<Question>`	?
`<Newline>`	^J	`<Rparen>`	)	`<Query>`	?
`<Linefeed>`	^J	`<Langle>`	<	`<Caret>`	^
`<Tab>`	^I	`<Rangle>`	>	`<Dollar>`	$
`<Bell>`	^G	`<LSquare>`	[	`<Bang>`	!
`<Backspace>`	^H	`<RSquare>`	]	`<Exclamation>`	!
`<FormFeed>`	^L	`<Lbracket>`	[	`<Quote>`	'
`<Esc>`	^[	`<Rbracket>`	]	`<SQuote>`	'
`<Escape>`	^[	`<Dot>`	.	`<DQuote>`	"
`<Null>`	^@				

Figure 4.3: Character mnemonics in regular expressions.

In a regular expression, the percent ('%') character quotes the next character, removing any special meaning that character may have. For example, the expression `x%+` refers to the string 'x+', whereas the pattern `x+` refers to the set that contains 'x', 'xx', 'xxx', and so on.

You can also quote characters by enclosing them in angle brackets. The expression `x<+>` refers to the string 'x+', the same as `x%+`. In place of the character itself, you can provide the name of the character inside the angle brackets. Figure 4.3 lists some of the character names Epsilon recognizes; you can also use any character name in the Unicode standard, such as `<Superscript two>`.

To search for the NUL character (the character with ASCII code 0), you must use the expression `<Nul>`, because an actual NUL character may not appear in a regular expression.

Instead of the character's name, you can provide its numeric value using the notation `<#`*number*`>`. The sequence `<#`*number*`>` denotes the character with ASCII code *number*. For example, the pattern `<#0>` provides another way to specify the NUL character, and the pattern `abc<#10>+` specifies the set of strings that begin with 'abc' and end with one or more newline characters (newline has ASCII value 10). You can enter the value in hexadecimal, octal, or binary by prefixing the number with '0x', '0o', or '0b', respectively. For example, `<#32>`, `<#0x20>`, `<#0o40>`, and `<#0b100000>` all yield a ⟨Space⟩ character (ASCII code 32).

Character Classes

In place of any letter, you can specify a *character class*. A character class consists of a sequence of characters between square brackets. For example, the character class [adef] stands for any of the following characters: 'a', 'd', 'e', or 'f'.

In place of a letter in a character class, you can specify a range of characters using a hyphen: the character class [a-m] stands for the characters 'a' through 'm', inclusively. The class [ae-gr] stands for the characters 'a', 'e', 'f', 'g', or 'r'. The class [a-zA-Z0-9] stands for any alphanumeric character.

To specify the complement of a character class, put a caret as the first character in the class. Using the above examples, the class [^a-m] stands for any character other than 'a' through 'm', and the class [^a-zA-Z0-9] stands for any non-alphanumeric character. Inside a character class, only ^ and - have special meaning. All other characters stand for themselves, including plus, star, question mark, etc.

If you need to put a right square bracket character in a character class, put it immediately after the opening left square bracket, or in the case of an inverted character class, immediately after the caret. For example, the class []x] stands for the characters ']' or 'x', and the class [^]x] stands for any character other than ']' or 'x'.

To include the hyphen character - in a character class, it must be the first character in the class, except for ^ and]. For example, the pattern [^]-q] matches any character except], -, or q.

Any regular expression you can write with character classes you can also write without character classes. But character classes sometimes let you write much shorter regular expressions.

The period character (outside a character class) represents any character except a ⟨Newline⟩. For example, the pattern a.c matches any three-character sequence on a single line where the first character is 'a' and the last is 'c'.

You can also specify a character class using a variant of the angle bracket syntax described in the previous section for entering special characters. The expression <Comma|Period|Question> represents any one of those three punctuation characters. The expression <a-z|A-Z|?> represents either a letter or a question mark, the same as [a-zA-Z]|<?>, for example. The expression <^Newline> represents any character except newline, just as the period character by itself does.

You can also use a few character class names that match some common sets of characters.

Class	Meaning
<digit>	A digit, 0 to 9.
<alpha>	A letter, according to isalpha().
<alphanum>	Either of the above.
<word>	All of the above, plus the _ character.
<hspace>	The same as <Space|Tab>.
<wspace>	The same as <Space|Tab|Newline>.
<ascii>	An ASCII character, one with a code below 128.
<any>	Any character including <Newline>.

Figure 4.4: Character Class Names

You can match all characters with a particular Unicode property, using the syntax <p:hex-digit>. After the p: part, you can put the name of a binary property as in p:ASCIIHexDigit, a script name as in p:Cyrillic, or a category name as in p:Zs or p:L. Or you can put the name of an enumerated property, an equal sign, and a value for that property, like p:block=Dingbats or p:Line_break=Alphabetic. Case

isn't significant in these names, and certain characters like hyphen and underscore are ignored in property names.

You can combine character classes using addition, subtraction, or intersection. Addition means a matching character can be in either of two classes, as in `<alpha|digit>` to match either alphabetic characters or digits. Intersection means a matching character must be a member of both classes, as in `<p:HexDigit&p:numeric-type=decimal>`, which matches characters with the HexDigit binary Unicode property that also have a Numeric-Type property of Decimal. Subtraction means a matching character must be a member of one class but not another, as in `<p:currency-symbol&!dollar sign&!cent sign>` which matches all characters with the Currency-Symbol property except for the dollar sign and cent sign characters.

More precisely, we can say that inside the angle brackets you can put one or more character "rules", each separated from the next by either a vertical bar `|` to add the rules together or `&` to intersect the rules. Any rule may have a `!` before it to invert that one rule, or you can put a `^` just after the opening `<` to invert the entire expression and match its complement.

Each character rule may be a character specification or a range, a character class name from the table above, or a Unicode property specification using the `p:` syntax above. A range means two character specifications with a hyphen between them. And a character specification means either the name of a character, or `#` and the numeric code for a character, or the character itself (for any character except `>`, `|`, `-`, or ⟨Nul⟩).

Separately, Epsilon recognizes the syntax `<h:0d 0a 45>` as a shorthand to search for a series of characters by their hexadecimal codes. This example is equivalent to the pattern `<#0x0d><#0x0a><#0x45>`.

Regular Expression Examples

- The pattern `if|else|for|do|while|switch` specifies the set of statement keywords in C and EEL.

- The pattern `c[ad]+r` specifies strings like 'car', 'cdr', 'caadr', 'caaadar'. These correspond to compositions of the car and cdr Lisp operations.

- The pattern `c[ad][ad]?[ad]?[ad]?r` specifies the strings that represent up to four compositions of car and cdr in Lisp.

- The pattern `[a-zA-Z]+` specifies the set of all sequences of 1 or more letters. The character class part denotes any upper- or lower-case letter, and the plus operator specifies one or more of those.

  Epsilon's commands to move by words accomplish their task by performing a regular expression search. They use a pattern similar to `[a-zA-Z0-9_]+`, which specifies one or more letters, digits, or underscore characters. (The actual pattern includes national characters as well.)

- The pattern `(<Newline>|<Return>|<Tab>|<Space>)+` specifies nonempty sequences of the whitespace characters newline, return, tab, and space. You could also write this pattern as `<Newline|Return|Tab|Space>+` or as `<Wspace|Return>+`, using a character class name.

- The pattern `/%*.*%*/` specifies a set that includes all 1-line C-language comments. The percent character quotes the first and third stars, so they refer to the star character itself. The middle star applies to the period, denoting zero or more occurrences of any character other than newline. Taken together then, the pattern denotes the set of strings that begin with "slash star", followed by any number of non-newline characters, followed by "star slash". You can also write this pattern as `/<Star>.*<Star>/`.

- The pattern `/%*(.|<Newline>)*%*/` looks like the previous pattern, except that instead of '.', we have `(.|<Newline>)`. So instead of "any character except newline", we have "any character except

newline, or newline", or more simply, "any character at all". This set includes all C comments, with or without newlines in them. You could also write this as `/%*<Any>*%*/` instead.

- The pattern `<^digit|a-f>` matches any character except of one these: 0123456789abcdef.

- The pattern `<alpha&!r&!x-z&!p:softdotted>` matches all Latin letters except R, X, Y, Z, I and J (the latter two because the Unicode property SoftDotted, indicating a character with a dot that can be replaced by an accent, matches I and J). It also matches all non-Latin Unicode letters that don't have this property.

AN ADVANCED EXAMPLE.

Let's build a regular expression that includes precisely the set of legal strings in the C programming language. All C strings begin and end with double quote characters. The inside of the string denotes a sequence of characters. Most characters stand for themselves, but newline, double quote, and backslash must appear after a "quoting" backslash. Any other character may appear after a backslash as well.

We want to construct a pattern that generates the set of all possible C strings. To capture the idea that the pattern must begin and end with a double quote, we begin by writing

"*something*"

We still have to write the *something* part, to generate the inside of the C strings. We said that the inside of a C string consists of a sequence of characters. The star operator means "zero or more of something". That looks promising, so we write

"(*something*)*"

Now we need to come up with a *something* part that stands for an individual character in a C string. Recall that characters other than newline, double quote, and backslash stand for themselves. The pattern `<^Newline|"|\>` captures precisely those characters. In a C string, a "quoting" backslash must precede the special characters (newline, double quote, and backslash). In fact, a backslash may precede any character in a C string. The pattern `\(.|<Newline>)` means, precisely "backslash followed by any character". Putting those together with the alternation operator (`|`), we get the pattern `<^Newline|"|\>|\(.|<Newline>)` which generates either a single "normal" character or any character preceded by a backslash. Substituting this pattern for the *something* yields

"(<^Newline|"|\>|\(.|<Newline>))*"

which represents precisely the set of legal C strings. In fact, if you type this pattern into a regex-search command (described below), Epsilon will find the next C string in the buffer.

Searching Rules

Thus far, we have described regular expressions in terms of the abstract set of strings they generate. In this section, we discuss how Epsilon uses this abstract set when it does a regular expression search.

When you tell Epsilon to perform a forward regex search, it looks forward through the buffer for the first occurrence in the buffer of a string contained in the generated set. If no such string exists in the buffer, the search fails.

There may exist several strings in the buffer that match a string in the generated set. Which one qualifies as the first one? By default, Epsilon picks the string in the buffer that begins before any of the others. If there exist two or more matches in the buffer that begin at the same place, Epsilon by default picks

the longest one. We call this a first-beginning, longest match. For example, suppose you position point at the beginning of the following line,

```
When to the sessions of sweet silent thought
```

then do a regex search for the pattern `s[a-z]*`. That pattern describes the set of strings that start with 's', followed by zero or more letters. We can find quite a few strings on this line that match that description. Among them:

```
When to the sessions of sweet silent thought
When to the sessions of sweet silent thought
When to the sessions of sweet silent thought
When to the sessions of sweet silent thought
When to the sessions of sweet silent thought
When to the sessions of sweet silent thought
```

Here, the underlined sections indicate portions of the buffer that match the description "s followed by a sequence of letters". We could identify 31 different occurrences of such strings on this line. Epsilon picks a match that begins first, and among those, a match that has maximum length. In our example, then, Epsilon would pick the following match:

```
When to the sessions of sweet silent thought
```

since it begins as soon as possible, and goes on for as long as possible. The search would position point after the final 's' in 'sessions'.

In addition to the default first-beginning, longest match searching, Epsilon provides three other regex search modes. You can specify first-beginning or first-ending searches. For each of these, you can specify shortest or longest match matches. Suppose, with point positioned at the beginning of the following line

```
I summon up remembrance of things past,
```

you did a regex search with the pattern `m.*c|I.*t`. Depending on which regex mode you chose, you would get one of the four following matches:

```
I summon up remembrance of things past,       (first-ending shortest)
I summon up remembrance of things past,        (first-ending longest)
I summon up remembrance of things past,         (first-beginning shortest)
I summon up remembrance of things past,         (first-beginning longest)
```

By default, Epsilon uses first-beginning, longest matching. You can include directives in the pattern itself to tell Epsilon to use one of the other techniques. If you include the directive `<Min>` anywhere in the pattern, Epsilon will use shortest-matching instead of longest-matching. Putting `<FirstEnd>` selects first-ending instead of first-beginning. You can also put `<Max>` for longest-matching, and `<FirstBegin>` for first-beginning. These last two might come in handy if you've changed Epsilon's default regex mode. The sequences `<FE>` and `<FB>` provide shorthand equivalents for `<FirstEnd>` and `<FirstBegin>`, respectively. As an example, you could use the following patterns to select each of the matches listed in the previous example:

```
<FE><Min>m.*c|I.*t                                                    (first-ending shortest)
<FE><Max>m.*c|I.*t          or      <FE>m.*c|I.*t               (first-ending longest)
<FB><Min>m.*c|I.*t          or      <Min>m.*c|I.*t              (first-beginning shortest)
<FB><Max>m.*c|I.*t          or      m.*c|I.*t                   (first-beginning longest)
```

You can change Epsilon's default regex searching mode. To make Epsilon use, by default, first-ending searches, set the variable `regex-shortest` to a nonzero value. To specify first-ending searches, set the variable `regex-first-end` to a nonzero value. (Examples of regular expression searching in this documentation assume the default settings.)

When Epsilon finds a regex match, it sets point to the end of the match. It also sets the variables `matchstart` and `matchend` to the beginning and end, respectively, of the match. You can change what Epsilon considers the end of the match using the '!' directive. For example, if you searched for 'I s!ought' in the following line, Epsilon would match the underlined section:

```
I sigh the lack of many a thing I sought,
```

Without the '!' directive, the match would consist of the letters "I sought", but because of the '!' directive, the match consists of only the indicated section of the line. Notice that the first three characters of the line also consist of 'I s', but Epsilon does not count that as a match. There must first exist a complete match in the buffer. If so, Epsilon will then set point and `matchend` according to any '!' directive.

OVERGENERATING REGEX SETS.

You can use Epsilon's regex search modes to simplify patterns that you write. You can sometimes write a pattern that includes more strings than you really want, and rely on a regex search mode to cut out strings that you don't want.

For example, recall the earlier example of `/%*(.|<Newline>)*%*/`. This pattern generates the set of all strings that begin with `/*` and end with `*/`. This set includes all the C-language comments, but it includes some additional strings as well. It includes, for example, the following illegal C comment:

```
/* inside /* still inside */ outside */
```

In C, a comment begins with `/*` and ends with the *very next* occurrence of `*/`. You can effectively get that by modifying the above pattern to specify a first-ending, longest match, with `<FE><Max>/%*(.|<Newline>)*%*/`. It would match:

```
/* inside /* still inside */ outside */
```

In this example, you could have written a more complicated regular expression that generated precisely the set of legal C comments, but this pattern proves easier to write.

Regular Expression Assertions

You can force Epsilon to reject any potential match that does not line up appropriately with a line boundary, by using the '^' and '$' assertions. A '^' assertion specifies a beginning-of-line match, and a '$' assertion specifies an end-of-line match. For example, if you search for `^new|waste` in the following line, it would match the indicated section:

```
And with old woes new wail my dear times's waste;
```

Even though the word 'new' occurs before 'waste', it does not appear at the beginning of the line, so Epsilon rejects it.

Other assertions use Epsilon's angle-bracket syntax. Like the assertions ^ and $, these don't match any specific characters, but a potential match will be rejected if the assertion isn't true at that point in the pattern.

Assertion	Meaning
^	At the start of a line.
$	At the end of a line.
<bob> or <bof>	At the start of the buffer.
<eob> or <eof>	At the end of the buffer.

For example, searching for <bob>sometext<eob> won't succeed unless the buffer contains only the eight character string sometext.

You can create new assertions from character classes specified with the angle bracket syntax by adding [,] or / at the start of the pattern.

Assertion	Meaning
<[*class*	The next character matches *class*, the previous one does not.
<]*class*	The previous character matches *class*, the next one does not.
</*class*	Either of the above.

The *class* in the above syntax is a |-separated or &-separated list of one or more single characters, character names like Space or Tab, character numbers like #32 or #9, ranges of any of these, character class names like Word or Digit, or Unicode property specifications. See page 58 for details on character classes.

For example, </word> matches at a word boundary, and <]word> matches at the end of a word. The pattern <]0-9|a-f> matches at the end of a run of hexadecimal digits. The pattern (cat|[0-9])</digit>(dog|[0-9]) matches cat3 or 4dog, but not catdog or 42. The pattern <[p:cyrillic> matches at the start of a run of Cyrillic characters.

COLOR CLASS ASSERTIONS.

Another type of assertion matches based on the next character's color class for syntax highlighting. <^c:*comment>printf finds uses of printf that aren't commented out. <[c:c-string>" finds " characters that start a string in C mode, ignoring those that end it, or appear quoted inside it, or in comments or other places.

The text after the c: is a simple filename-style pattern that will be matched against the name of the color class: * matches zero or more characters, ? matches any single character, and simple ranges with [] are allowed. A character with no syntax highlighting applied will match the name "none". This type of assertion may start with ^ to invert the matching rules, or with /, [or] to match color boundaries.

To apply more than one assertion to a character, put them in sequence. <^c:c-string><^c:*comment>printf finds instances of printf that are in neither strings nor comments.

You can use the set-color command to see the color class names Epsilon uses.

In extension language code, use the do_color_searching() subroutine if your regular expression might include syntax highlighting assertions, which ensures the buffer's syntax highlighting is up to date.

Regular Expression Commands

You can invoke a forward regex search with the Ctrl-Alt-S key, which runs the command regex-search. The Ctrl-Alt-R key invokes a reverse incremental search. You can also enter regular expression mode from any search prompt by typing Ctrl-T to that prompt. For example, if you press Ctrl-S to invoke

incremental-search, pressing Ctrl-T causes it to enter regular expression mode. See page 38 for a description
of the searching commands.

The key Alt-* runs the command regex-replace. This command works like the command query-replace,
but interprets its search string as a regular expression.

In the replacement text of a regex replace, the # character followed by a digit *n* has a special meaning in
the replacement text. Epsilon finds the *n*th parenthesized expression in the pattern, counting left parentheses
from 1. It then substitutes the match of this subpattern for the #*n* in the replacement text. For example,
replacing

```
([a-zA-Z0-9_]+) = ([a-zA-Z0-9_]+)
```

with

```
#2 := #1
```

changes

```
variable = value;
```

to

```
value := variable;
```

If #0 appears in the replacement text, Epsilon substitutes the entire match for the search string. To
include the actual character # in a replacement text, use ##. In a search pattern, you can follow the open
parenthesis with ?: to tell Epsilon not to count it for replacement purposes; that pair of parentheses will
only be used for grouping.

The replacement text can use the syntax #U to force the rest of the replacement to uppercase (including
text substituted from the match using #1 syntax). Using #L or #C forces the remaining text to lowercase, or
capitalizes it, respectively. Using #E marks the end of such case modifications; the following replacement
text will be substituted as-is. For instance, searching for "(<word>+) by (<word>+)" and replacing it
with "#L#2#E By #U#1" will change the match "Two by Four" into "four By TWO".

When you don't use the above syntax, replacing preserves the case of each match according to specific
rules. See the `replace-by-case` variable for details.

Characters other than # in the replacement text have no special meaning. To enter special characters,
type a Ctrl-Q before each. Type Ctrl-Q Ctrl-J to include a ⟨Newline⟩ character in the replacement text. Or
specific characters in the replacement text by name, using the syntax #<Newline>, or by number, such as
#<#0x221a> for the Unicode square root character.

Summary:	Ctrl-Alt-S	regex-search
	Ctrl-Alt-R	reverse-regex-search
	Alt-*	regex-replace

4.3.8   Rearranging

Sorting

Epsilon provides several commands to sort buffers, or parts of buffers.

The sort-buffer command lets you sort the lines of the current buffer. The command asks for the name
of a buffer in which to place the sorted output. The sort-region command sorts the part of the current buffer

between point and mark, in place. The commands reverse-sort-buffer and reverse-sort-region operate like the above commands, but reverse the sorting order.

By default, all the sorting commands sort the lines by considering all the characters in the line. If you prefix a numeric argument of n to any of these commands, they will compare lines starting at column n.

When comparing lines of text during sorting, Epsilon normally folds lower case letters to upper case before comparison, if the `case-fold` variable has a nonzero value. If the `case-fold` variable has a value of 0, Epsilon compares characters as-is. However, setting the buffer-specific `sort-case-fold` variable to 0 or 1 overrides the `case-fold` variable, for sorting purposes. By default, `sort-case-fold` has a value of 2, which means to defer to `case-fold`.

Summary: sort-buffer
sort-region
reverse-sort-buffer
reverse-sort-region

Transposing

Epsilon has commands to transpose characters, words, and lines. To transpose the words before and after point, use the Alt-T command. This command leaves undisturbed any non-word characters between the words. Point moves between the words. The Ctrl-X Ctrl-T command transposes the current and previous lines and moves point between them.

The Ctrl-T command normally transposes the characters before and after point. However, at the start of a line it transposes the first two characters on the line, and at the end of a line it transposes the last two. On a line with one or no characters, it does nothing.

Summary: Ctrl-T   transpose-characters
Alt-T   transpose-words
Ctrl-X Ctrl-T   transpose-lines

Formatting Text

Epsilon has some commands that make typing manuscript text easier.

You can change the right margin, or *fill column*, using the Ctrl-X F command. By default, it has a value of 70. With a numeric argument, the command sets the fill column to that column number. Otherwise, this command tells you the current value of the fill column and asks you for a new value. If you don't provide a new value but instead press the ⟨Enter⟩ key, Epsilon will use the value of point's current column. For example, you can set the fill column to column 55 by typing Ctrl-U 55 Ctrl-X F. Alternatively, you can set the fill column to point's column by typing Ctrl-X F ⟨Enter⟩. The buffer-specific variable `margin-right` stores the value of the fill column. To set the default value for new buffers you create, use the set-variable command on F8 to set the default value of the `margin-right` variable. (See the `c-fill-column` variable for the C mode equivalent.) A file's contents can specify a particular fill column; see page 103.

In *auto fill mode*, you don't have to worry about typing ⟨Enter⟩'s to go to the next line. Whenever a line gets too long, Epsilon breaks the line at the appropriate place if needed. The auto-fill-mode command enables or disables auto filling (word wrap) for the current buffer. With a numeric argument of zero, it turns auto filling off; with a nonzero numeric argument, it turns auto filling on. With no numeric argument, it

toggles auto filling. During auto fill mode, Epsilon shows the word "Fill" in the mode line. The buffer-specific variable `fill-mode` controls filling. If it has a nonzero value, filling occurs. To make Epsilon always use auto fill mode, you can use the set-variable command to set the default value of `fill-mode`.

In some language modes, Epsilon uses a special version of auto-fill mode that typically only fills text in certain types of comments. See page 82 for details.

Epsilon normally indents new lines it inserts via auto fill mode so they match the previous line. The buffer-specific variable `auto-fill-indents` controls whether or not Epsilon does this. Epsilon indents these new lines only if `auto-fill-indents` has a nonzero value. Set the variable to 0 if you don't want this behavior.

During auto filling, the normal-character command first checks to see if the line extends past the fill column. If so, the extra words automatically move down to the next line.

The ⟨Enter⟩ key runs the command enter-key, which behaves like normal-character, but inserts a newline instead of the character that invoked it. Epsilon binds this command to the ⟨Enter⟩ key, because Epsilon uses the convention that Ctrl-J's separate lines, but the keyboard has the ⟨Enter⟩ key yield a Ctrl-M. In overwrite mode, the ⟨Enter⟩ key simply moves to the beginning of the next line.

The Alt-Q command fills the current paragraph. The command fills each line by moving words between lines as necessary, so the lines but the last become as long as possible without extending past the fill column. If the screen shows a highlighted region, the command fills all paragraphs in the region. The fill-region command fills all paragraphs in the region between point and mark, whether or not the region is highlighted.

If you give a numeric prefix argument of five or less to the above filling commands, they unwrap lines in a paragraph, removing all line breaks. Alt-2 Alt-Q is one quick way to unwrap the current paragraph. With a numeric argument greater than 5, the paragraph is filled using that value as a temporary right margin. (Note that C mode places a different fill command on Alt-Q, and it interprets an argument to mean "fill using the current column as a right margin".)

Alt-Shift-Q runs the fill-indented-paragraph command, which fills the current paragraph as above but also tries to preserve any indentation before each line of the paragraph. With a numeric argument, it fills the paragraph using the current column as the right margin, instead of the `margin-right` variable.

Summary: Ctrl-X F   set-fill-column
                Alt-q   fill-paragraph
                Alt-Shift-Q   fill-indented-paragraph
                                                      fill-region
                                                      auto-fill-mode
                ⟨Enter⟩   enter-key

4.3.9 Indenting Commands

Epsilon can help with indenting your program or other text. The ⟨Tab⟩ key runs the indent-previous command, which makes the current line start at the same column as the previous non-blank line. Specifically, if you invoke this command with point in or adjacent to a line's indentation, indent-previous replaces that indentation with the indentation of the previous non-blank line. If point's indentation exceeds that of the previous non-blank line, or if you invoke this command with point outside of the line's indentation, this command simply inserts a ⟨Tab⟩. See page 88 for information on changing the width of a tab.

Epsilon can automatically indent for you when you press ⟨Enter⟩. Setting the buffer-specific variable `auto-indent` nonzero makes Epsilon do this. The way Epsilon indents depends on the current mode. For

example, C mode knows how to indent for C programs. In Epsilon's default mode, fundamental mode, Epsilon indents like indent-previous if you set `auto-indent` nonzero.

When Epsilon automatically inserts new lines for you in auto fill mode, it looks at a different variable to determine whether to indent these new lines. Epsilon indents in this case only if the buffer-specific variable `auto-fill-indents` has a nonzero value.

The Alt-M key moves point to the beginning of the text on the current line, just past the indentation.

The indent-under command functions like indent-previous, but each time you invoke it, it indents more, to align with the next word in the line above. In detail, it goes to the same column in the previous non-blank line, and looks to the right for the end of the next region of spaces and tabs. It indents the current line to that column after removing spaces and tabs from around point.

The indent-rigidly command, bound to Ctrl-X Ctrl-I (or Ctrl-X ⟨Tab⟩), changes the indentation of each line between point and mark by a fixed amount provided as a numeric argument. For instance, Ctrl-U 8 Ctrl-X Ctrl-I moves all the lines to the right by eight spaces. With no numeric argument, lines move to the right by the buffer's tab size (default 8; see page 88), and with a negative numeric argument, lines move to the left. So, for example, Ctrl-U -1000 Ctrl-X Ctrl-I should remove all the indentation from the lines between point and mark.

If you highlight a region before pressing ⟨Tab⟩ (or any key that runs one of the commands indent-previous, indent-under, or do-c-indent), Epsilon indents all lines in the region by one tab stop, by calling the indent-rigidly command. You can provide a numeric argument to specify how much indentation you want.

The Shift-⟨Tab⟩ key moves the cursor back to the previous tab stop. But if you highlight a region before pressing it, it will remove one tab stop's worth of indentation. (See the `resize-rectangle-on-tab` variable if you want these keys to instead change the region's shape without moving text.)

The indent-region command, bound to Ctrl-Alt-\, works similarly. It goes to the start of each line between point and mark and invokes the command bound to ⟨Tab⟩. If the resulting line then contains only spaces and tabs, Epsilon removes them.

You can set up Epsilon to automatically reindent text when you yank it. Epsilon will indent like indent-region. By default, Epsilon does this only for C mode (see the `reindent-after-c-yank` variable).

To determine whether to reindent yanked text, the yank command first looks for a variable whose name is derived from the buffer's mode as it appears in the mode line: `reindent-after-c-yank` for C mode buffers, `reindent-after-html-yank` for HTML mode buffers, and so forth. If there's no variable by that name, Epsilon uses the `reindent-after-yank` variable instead. Instead of a variable, you can write an EEL function with the same name; Epsilon will call it and use its return value. See the description of `reindent-after-yank` for details on what different values do.

The Alt-S command horizontally centers the current line between the first column and the fill column by padding the left with spaces and tabs as necessary. Before centering the line, the command removes spaces and tabs from the beginning and end of the line.

With any of these commands, Epsilon indents by inserting as many tabs as possible without going past the desired column, and then inserting spaces as necessary to reach the column. You can set the size of a tab by setting the `tab-size` variable. Set the `soft-tab-size` variable if you want Epsilon to use one setting for displaying existing tab characters, and a different one for indenting.

If you prefer, you can make Epsilon indent using only spaces. The buffer-specific variable `indent-with-tabs` controls this behavior. Set it to 0 using set-variable to make Epsilon use only spaces when inserting indentation.

The untabify-region command on Ctrl-X Alt-I changes all tab characters between point and mark to the number of spaces necessary to make the buffer look the same. The tabify-region command on Ctrl-X Alt-⟨Tab⟩ does the reverse. It looks at all runs of spaces and tabs, and replaces each with tabs and spaces to

occupy the same number of columns. The commands tabify-buffer and untabify-buffer are similar, but operate on the entire buffer, instead of just the region.

Summary: Alt-M   to-indentation
                ⟨Tab⟩   indent-previous
                Shift-⟨Tab⟩   back-to-tab-stop
                Ctrl-Alt-I   indent-under
                Ctrl-X ⟨Tab⟩   indent-rigidly
                Ctrl-Alt-\   indent-region
                Alt-S   center-line
                Ctrl-X Alt-⟨Tab⟩   tabify-region
                Ctrl-X Alt-I   untabify-region
                                                    tabify-buffer
                                                    untabify-buffer

4.3.10   Hex Mode

The hex-mode command creates a second buffer that shows a hex listing of the original buffer. You can edit this buffer, as explained below. Press q when you're done, and Epsilon will return to the original buffer, offering to apply your changes.

A hex digit (0-9, a-f) in the left-hand column area moves in the hex listing to the new location.

A hex digit (0-9, a-f) elsewhere in the hex listing modifies the listing.

q quits hex mode, removing the hex mode buffer and returning to the original buffer. Epsilon will first offer to apply your editing changes to the original buffer.

⟨Tab⟩ moves between the columns of the hex listing.

s or r searches by hex bytes. Type a series of hex bytes, like 0a 0d 65, and Epsilon will search for them. S searches forward, R in reverse.

Ctrl-S and Ctrl-R temporarily toggle to the original buffer so you can search for literal text. When the search ends, they move to the corresponding place in the hex listing.

t toggles between the original buffer and the hex mode buffer, going to the corresponding position.

prompts for a new character value and overwrites the current character with it. You can use any of these formats: 'A', 65, 0x41 (hex), 0b1100101 (binary), 0o145 (octal).

n or p move to the next or previous line.

g prompts for an offset in hexadecimal, then goes there.

o toggles the hex overwrite submode, which changes how Epsilon interprets keys you type in the rightmost column of the hex listing. In overwrite mode, printable characters you type in the rightmost column overwrite the text there, instead of acting as hex digits or commands.

For instance, typing "3as" in the last column while in overwrite mode replaces the next three characters with the characters 3, a, and s. Outside overwrite mode, they replace the current character with one whose hex code is 3a, and then begin a search.

To use hex mode commands from overwrite mode, prefix them with a Ctrl-C character, such as Ctrl-C o to exit overwrite mode. Or move out of the rightmost column with ⟨Tab⟩ or other movement keys.

? shows help on hex mode.

Summary: hex-mode

4.4   Language Modes

When you use the find-file command to read in a file, Epsilon looks at the file's extension to see if it has a mode appropriate for editing that type of file. For example, when you read a .h file, Epsilon goes into C mode. Specifically, whenever you use find-file and give it a file name "foo.*ext*", after find-file reads in the file, it executes a command named "`suffix_ext`", if such a command exists. The find-file command constructs a subroutine name from the file extension to allow you to customize what happens when you begin editing a file with that extension.

For example, if you want to enter C mode automatically whenever you use find-file on a ".x" file, you simply create a command (a keyboard macro would do) called "`suffix_x`", and have that command call c-mode, or even better, an existing `suffix_` function. One way is to add a line like this to your einit.ecm file (see page 131):

```
(define-macro "suffix-x" "<!suffix-c>")
```

For another example, you can easily stop Epsilon from automatically entering C mode on a ".h" file by using the delete-name command to delete the subroutine "`suffix-h`". (You can interchange the - and _ characters in Epsilon command names.) Or define a suffix-h macro so it calls the fundamental-mode command in your einit.ecm file, as above.

Epsilon also has various features that are useful in many different language modes. See the description of tagging on page 43 and the section starting on page 80.

In addition to the language-specific modes described in the following sections, Epsilon includes modes that support various Epsilon features. For example, the buffer listing generated by the bufed command on Ctrl-X Ctrl-B is actually in an Epsilon buffer, and that buffer is in Bufed mode.

Many language modes will call a hook function if you've defined one. For example, C mode tries to call a function named `c_mode_hook()`. A hook function is a good place to customize a mode by setting buffer-specific variables. It can be a keyboard macro or a function written in EEL, and it will be called whenever Epsilon loads a file that should be in the specified mode.

The fundamental-mode command removes changes to key bindings made by modes such as C mode, Dired mode, or Bufed mode. You can configure Epsilon to highlight matching parentheses and other delimiters in fundamental mode; see the `fundamental-auto-show-delim-chars` variable.

Also see page 101 to customize the list of file types shown in File/Open and similar dialogs in Epsilon for Windows.

Summary: fundamental-mode

4.4.1   Asm Mode

Epsilon automatically enters Asm mode when you read a file with an extension of .asm, .inc, .al, .mac, .ah, or .asi. In Asm mode, Epsilon does appropriate syntax highlighting, tagging, and commenting. The compile-buffer command uses the `compile-asm-cmd` variable in this mode.

Summary: asm-mode

4.4.2   Batch Mode

Epsilon automatically enters Batch mode when you read a file with an extension of .bat, .cmd, or .btm. In Batch mode, Epsilon does appropriate syntax highlighting.

Summary: batch-mode

4.4.3   C Mode

The c-mode command puts the current buffer in C mode. C mode provides smart indenting for programs written in C, C++, C#, Java, Epsilon's extension language EEL, and other C-like languages. Pressing ⟨Enter⟩ or ⟨Tab⟩ examines previous lines to find the correct indentation. Epsilon supports several common styles of indentation, controlled by some extension language variables.

The Closeback variable controls the position of the closing brace:

Closeback = 0;   *Closeback = 1;*

```
if (foo){                                 if (foo){
    bar();                                    bar();
    baz();                                    baz();
    }                                     }
```

By placing the opening brace on the following line, you may also use these styles:

Closeback = 0;   *Closeback = 1;*

```
if (foo)                                  if (foo)
    {                                     {
    bar();                                    bar();
    baz();                                    baz();
    }                                     }
```

Closeback by default has a value of 1.

Use the Topindent variable to control the indentation of top-level statements in a function:

Topindent = 0;   *Topindent = 1;*

```
foo()                                     foo()
{                                         {
if (bar)                                      if (bar)
    baz();                                        baz();
}                                         }
```

Topindent by default has a value of 1.

The `Matchdelim` variable controls whether typing),], or } displays the corresponding (, [, or { using the show-matching-delimiter command. The `Matchdelim` variable normally has a value of 1, which means that Epsilon shows matching delimiters. You can change these variables as described on page 127.

In C mode, the ⟨Tab⟩ key reindents the current line if pressed with point in the current line's indentation. ⟨Tab⟩ just inserts a tab if pressed with point somewhere else, or if pressed two or more times successively. If you set the variable `c-tab-always-indents` to 1, then the ⟨Tab⟩ key will reindent the current line, regardless of your position on the line. If you press it again, it will insert another tab.

When you yank text into a buffer in C mode, Epsilon automatically reindents it. This is similar to the "smart paste" feature in some other editors. You can set the variable `reindent-after-c-yank` to zero to disable this behavior. Epsilon doesn't normally reindent comments when yanking; set the `reindent-c-comments` and `reindent-one-line-c-comments` variables to change that. Also see the `reindent-c-preprocessor-lines` variable.

By default, Epsilon uses the value of the buffer-specific `tab-size` variable to determine how far to indent. For example, if the tab size has a value of 5, Epsilon will indent the line following an `if` statement five additional columns.

If you want the width of a tab character in C mode buffers to be different than in other buffers, set the variable `c-tab-override` to the desired value. C mode will change the buffer's tab size to the specified number of columns. The `eel-tab-override` variable does the same in EEL buffers (which use a variation of C mode). Also see the description of file variables on page 103 for a way in which individual files can indicate they should use a particular tab size.

If you want to use one value for the tab size and a different one for C indentation, set the buffer-specific `c-indent` variable to the desired indentation using the set-variable command. When `c-indent` has a value of zero, as it has by default, Epsilon uses the `tab-size` variable for its indentation. (Actually, the ⟨Tab⟩ key in C mode doesn't necessarily insert a tab when you press it two or more times in succession. Instead, it indents according to `c-indent`. If the tab size differs from the C indent, it may have to insert spaces to reach the proper column.)

In Java files, Epsilon uses the similar variable `java-indent` to set the column width of one level of indentation.

The `c-case-offset` variable controls the indentation of `case` statements. Normally, Epsilon indents them one level more than their controlling `switch` statements. Epsilon adds the value of this variable to its normal indentation, though. If you normally indent by 8 spaces, for example, and want `case` statements to line up with their surrounding `switch` statements, set `c-case-offset` to -8.

Similarly, the `c-access-spec-offset` variable controls the indentation of `public:`, `private:`, `protected:` (and, for C#, `internal:`) access specifiers.

The `c-label-indent` variable provides the indentation of lines starting with labels. Normally, Epsilon moves labels to the left margin.

Epsilon offsets the indentation of a left brace on its own line by the value of the variable `c-brace-offset`. For example, with a tab size of eight and default settings for other variables, a `c-brace-offset` of 2 produces:

```
if (a)
  {
        b();
  }
```

The variable `c-top-braces` controls how much Epsilon indents the braces of the top-level block of a function. By default, Epsilon puts these braces at the left margin. Epsilon indents pre-ANSI K&R-style

parameter declarations according to the variable `c-param-decl`. Epsilon indents parts of a top-level structure or union according to `c-top-struct`, and indents continuation lines outside of any function body according to `c-top-contin`. Continuation lines for classes and functions that use C++ inheritance syntax may be indented according to `c-align-inherit`. Additional C mode indentation variables that may be customized include `c-indent-after-extern-c` and `c-indent-after-namespace`.

By default, the C indenter tries to align continuation lines under parentheses and other syntactic items on prior lines. If Epsilon can't find anything on prior lines to align under, it indents continuation lines two levels more than the original line. (With default settings, Epsilon indents unalignable continuation lines 8 positions to the right of the original line.) Epsilon adds the value of the variable `c-contin-offset` to this indentation, though. If you want Epsilon to indent unalignable continuation lines ten columns less, set `c-contin-offset` to -10 (it's 0 by default).

If aligning the continuation line would make it start in a column greater than the value of the variable `c-align-contin-lines` (default 48), Epsilon won't align the continuation line. It will indent by two levels plus the value of `c-contin-offset`, as described above. Also see the `c-align-extra-space` variable for an adjustment Epsilon can make for continuation lines that would be indented exactly one level.

As a special case, setting the `c-align-contin-lines` to zero makes Epsilon never try to align continuation lines under syntactic features on prior lines. Epsilon will then indent all continuation lines by one level more than the original line (one extra tab, normally), plus the value of the variable `c-contin-offset`.

If the continuation line contains only a left parenthesis character (ignoring comments), Epsilon can align it with the start of the current statement if you set `c-align-open-paren` nonzero. If the variable is zero, it's aligned like other continuation lines.

Summary: c-mode
              C Mode only: ⟨Tab⟩   do-c-indent
              C Mode only: {   c-open
              C Mode only: }   c-close
              C Mode only: :   c-colon
              C Mode only: #   c-hash-mark
              C Mode only:),]   show-matching-delimiter

Other C mode Features

In C mode, the Alt-⟨Down⟩ and Alt-⟨Up⟩ keys move to the next or previous #if/#else/#endif preprocessor line. When starting from such a line, Epsilon finds the next/previous matching one, skipping over inner nested preprocessor lines. Alt-] and Alt-[do the same. Press Alt-i to display a list of the preprocessor conditionals that are in effect for the current line.

When the cursor is on a brace, bracket, or parenthesis, Epsilon will try to locate its matching brace, bracket, or parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-c-delimiters` to zero to disable this feature.

Press Alt-' to display a list of all functions and global variables defined in the current file. You can move to a definition in the list and press ⟨Enter⟩ and Epsilon will go to that definition, or press Ctrl-G to remain at the starting point. By default, this command skips over external declarations. With a prefix numeric argument, it includes those too. Also see the `list-which-definitions` variable.

Epsilon normally auto-fills text in block comments as you type, breaking overly long lines. See the `c-auto-fill-mode` variable. As with normal auto-fill mode (see page 65), use Ctrl-X F to set the right

margin for filling. Set the `c-fill-column` variable to change the default right margin in C mode buffers. Set `fill-c-comment-plain` nonzero if you want block comments to use only spaces instead of a `*` on successive lines.

You can manually refill the current paragraph in a block comment (or in a comment that follows a line of code) by pressing Alt-q. If you provide a numeric prefix argument to Alt-q, say by typing Alt-2 Alt-q, it will fill using the current column as the right margin.

Epsilon's tagging facility isn't specific to C mode, so it's described elsewhere (see page 43). But it's one of Epsilon's most useful software development features, so we mention it here too.

Whenever you use the find-file command to read in a file with one of the extensions .c, .h, .e, .y, .cpp, .cxx, .java, .inl, .hpp, .idl, .acf, .cs, .i, or .hxx, Epsilon automatically enters C mode. See page 69 for information on adding new extensions to this list, or preventing Epsilon from automatically entering C mode. For file names without a suffix, Epsilon examines their contents and guesses whether the file is C++, Perl, some other known type, or unrecognizable.

Summary:	C Mode only: Alt-], Alt-⟨Down⟩	forward-ifdef
	C Mode only: Alt-[, Alt-⟨Up⟩	backward-ifdef
	C Mode only: Alt-q	fill-comment
	Alt-'	list-definitions
	Alt-i	list-preprocessor-conditionals

4.4.4   Configuration File Mode

Epsilon automatically enters Conf mode when you read a file with an extension of .conf, or (under Unix only) when you read a non-binary file in the /etc directory. In Conf mode, Epsilon does some generic syntax highlighting, recognizing `#` and `;` as commenting characters, and highlighting name=value assignments. It also breaks and fills comments.

Summary: conf-mode

4.4.5   GAMS Mode

Epsilon automatically enters GAMS mode when you read a file with an extension of .gms or .set. In addition, if you set the `gams-files` variable nonzero, it recognizes .inc, .map, and .dat extensions. Epsilon also uses GAMS mode for files with an unrecognized extension that start with a GAMS `$title` directive. The GAMS language is used for mathematical programming.

In GAMS mode, Epsilon does syntax highlighting, recognizing GAMS strings and comments. The GAMS language permits a file to define its own additional comment character sequences, besides the standard `*` and `$ontext` and `$offtext`, and Epsilon recognizes most common settings for these.

When the cursor is on a bracket or parenthesis, Epsilon will try to locate its matching bracket or parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-gams-delimiters` to zero to disable this feature.

Summary: gams-mode

4.4.6   HTML and XML Modes

Epsilon automatically enters HTML mode when you read a file with an extension of .htm, .html, .shtml, .cfml, .cfm, .htx, .asp, .asa, .htt, .jsp, .prx, .asx, or .aspx. It uses XML mode when you read a file with an extension of .xml, .cdf, .osd, .wml, .xsl, .xst, .xsd, .xmp, .rdf, .svg, .sgml, or .sgm, or when a file identified as HTML has contents that suggest XML. Everything about HTML mode below also applies to XML mode.

In HTML mode, Epsilon does appropriate syntax highlighting (including embedded JavaScript or VBScript) and brace-matching. The commenting commands work too. The `html-auto-fill-mode` variable controls whether Epsilon automatically breaks long lines as you type. The value 3 makes it break HTML, but not language comments within any embedded scripting. The value 2 makes it break comments in scripting, but not HTML. The value 1 makes it do both, and 0 makes it do neither.

You can customize how Epsilon colors embedded scripting. Each of the variables in figure 4.5 may be set to 1 for Javascript-style coloring, 2 for VBScript-style coloring, 3 for PHP-style coloring, 4 to ignore those block start delimiters, or 0 for plain coloring of the block.

Variable	Default	Embedding Syntax
`html-asp-coloring`	JavaScript	`<% %>` (in HTML mode)
`html-php-coloring`	PHP	`<? ?>` (in HTML mode)
`xml-asp-coloring`	Ignore	`<% %>` (in XML mode)
`xml-php-coloring`	Ignore	`<? ?>` (in XML mode)
`html-php-coloring`	PHP	`<?php ?>`
`html-vbscript-coloring`	VBScript	`<script language=vbscript>`
`html-php-coloring`	PHP	`<script language=php>`
`html-javascript-coloring`	JavaScript	`<script language=javascript>`
`html-javascript-coloring`	JavaScript	`<script language=jscript>`
`html-javascript-coloring`	JavaScript	`<script language=ecmascript>`
`html-javascript-coloring`	JavaScript	`<jsp:declaration>`
`html-javascript-coloring`	JavaScript	`<jsp:expression>`
`html-javascript-coloring`	JavaScript	`<jsp:scriptlet>`
`html-other-coloring`	JavaScript	`<script language=`*other*`>`

Figure 4.5: Variables that control how Epsilon interprets embedded scripting

See the `html-display-definition` variable to control whether Epsilon displays the current function name or scripting type as you move about in the buffer.

When the cursor is on a < or > character, Epsilon will try to locate its matching > or < and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-html-delimiters` to zero to disable this feature.

When the cursor is at a start tag, you can press Alt-= to have Epsilon move to its matching end tag, and vice versa.

Summary: html-mode
                                                                     xml-mode
                          HTML modes only: Alt-=   html-find-matching-tag

4.4.7   Ini File Mode

Epsilon automatically enters Ini mode when you read a file with an extension of .ini or .sys. In Ini mode, Epsilon does appropriate syntax highlighting.

Summary: ini-mode

4.4.8   Makefile Mode

Epsilon automatically enters Makefile mode when you read a file named makefile (or Makefile, etc.) or with an extension of .mak or .mk. In Makefile mode, Epsilon does appropriate syntax highlighting, and can break and fill comments. The compile-buffer command uses the `compile-makefile-cmd` variable in this mode. Press Alt-i to display a list of the preprocessor conditionals that are in effect for the current line. (For this command, Epsilon assumes that a makefile uses Gnu Make syntax under Unix, and Microsoft or MKS makefile syntax elsewhere.)

Summary: makefile-mode
                    Makefile mode only: Alt-i   list-make-preprocessor-conditionals

4.4.9   Perl Mode

Epsilon automatically enters Perl mode when you read a file with an extension of .perl, .pm, .al, .ph, or .pl (or when you read a file with no extension that starts with a #! line mentioning Perl). The compile-buffer command uses the `compile-perl-cmd` variable in this mode.

Epsilon includes a perldoc command that you can use to read Perl documentation. It runs the command of the same name and displays the result in a buffer. You can double-click on a reference to another perldoc page, or press ⟨Enter⟩ to follow a reference at point, or press m to be prompted for another perldoc topic name.

Epsilon's syntax highlighting uses the `perl-comment` color for comments and POD documentation, the `perl-function` color for function names, and the `perl-variable` color for variable names.

Epsilon uses the `perl-constant` color for numbers, labels, the simple argument of an angle operator such as `<INPUT>`, names of imported packages, buffer text after `__END__` or `__DATA__`, here documents, format specifications (apart from any variables and comments within), and the operators `my` and `local`.

A here document can indicate that its contents should be syntax highlighted in a different language, by specifying a terminating string with an extension. At the moment the extensions .tex and .html are recognized. So for example a here document that begins with `<<"end.html"` will be colored as HTML.

Epsilon uses the `perl-string` color for string literals of all types (including regular expression arguments to `s///`, for instance). Interpolated variables and comments are colored appropriately whenever the string's context permits interpolation.

Epsilon uses the `perl-keyword` color for selected Perl operators (mostly those involved in flow control, like foreach or goto, or with special syntax rules, like tr or format), and modifiers like `/x` after regular expressions.

Perl mode's automatic indentation features use a modified version of C mode. See page 70 for information on customizing indentation. Perl uses a different set of customization variables whose names all start with `perl-` instead of `c-` but work the same as their C mode cousins. These include `perl-align-contin-lines`, `perl-brace-offset`, `perl-closeback`, `perl-contin-offset`, `perl-label-indent`, `perl-top-braces`, `perl-top-contin`, `perl-top-struct`, and `perl-topindent`. Set `perl-tab-override` if you want Epsilon to assume that tab characters in Perl files aren't always 8 characters wide. Set `perl-indent` if you want to use an indentation in Perl files that's not equal to one tab stop.

When the cursor is on a brace, bracket, or parenthesis, Epsilon will try to locate its matching brace, bracket, or parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-perl-delimiters` to zero to disable this feature.

When you yank blocks of text into a buffer in Perl mode, Epsilon can automatically reindent it. See the variable `reindent-after-perl-yank` to enable this behavior. Some Perl syntax is sensitive to indentation, and Epsilon's indenter may change the indentation, so you should examine yanked text to make sure it hasn't changed.

If you run Perl's debugger inside Epsilon's process buffer under Windows, the following environment variable settings are recommended:

```
set PERLDB_OPTS=ReadLine=0 ornaments=''
set EMACS=yes
```

Summary: perl-mode

                                                                      perldoc

4.4.10   PHP Mode

Epsilon automatically enters PHP mode when you read a file with an extension of .php, .php3, .php4, or .sphp. In PHP mode, Epsilon does appropriate syntax highlighting and similar tasks. PHP mode is almost identical to HTML mode. (See page 74.) In both, codes such as `<?` `?>` mark PHP scripting, and text outside of these markers is treated as HTML.

When the cursor is on a brace, bracket, or parenthesis in PHP code, Epsilon will try to locate its matching brace, bracket, or parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-php-delimiters` to zero to disable this feature.

When you use the indent-for-comment command to insert a comment, the `php-comment-style` variable controls which type of comment Epsilon inserts. The value 1 (the default) inserts `#`, the value 2 inserts `//`, and any other value inserts `/*`.

For customizing its indenting rules, PHP shares these variables with Perl mode: `perl-closeback`, `perl-topindent`, `perl-indent`, `perl-label-indent`, `perl-contin-offset`, `perl-align-contin-lines`, `perl-brace-offset`, `perl-top-braces`, `perl-top-contin`, and `perl-top-struct`.

Summary: php-mode

4.4.11 PostScript Mode

Epsilon automatically enters PostScript mode when you read a file with an extension of .ps or .eps, or if it contains a PostScript marker on its first line. In PostScript mode, Epsilon does appropriate syntax highlighting, recognizing text strings, comments, and literals like `/Name`. It also breaks and fills comment lines.

When the cursor is on a brace, bracket, or parenthesis, Epsilon will try to locate its matching brace, bracket, or parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-postscript-delimiters` to zero to disable this feature.

Summary: postscript-mode

4.4.12 Python Mode

Epsilon automatically enters Python mode when you read a file with an extension of .py. In Python mode, Epsilon does appropriate syntax highlighting. Tagging, comment filling, and other commenting commands are also available. Auto-indenting adds an extra level of indentation after a line ending with ":", and repeats the previous indentation otherwise.

Set the `python-indent` variable to alter the level of indentation Epsilon uses. Tab widths in Python files are normally set to 8, as required by Python language syntax rules, but you can set the `python-tab-override` variable to change this, or `python-indent-with-tabs` to change whether Python mode uses tabs for indenting, not purely spaces.

When the cursor is on a brace, bracket, or parenthesis, Epsilon will try to locate its matching brace, bracket, or parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-python-delimiters` to zero to disable this feature.

Set `compile-python-cmd` to modify the command line used by the compile-buffer command for Python buffer.

Summary: python-mode

4.4.13 Shell Mode

Epsilon automatically enters shell mode when you read a file with an extension of .sh, .csh, .ksh, .bash, .tcsh, or .zsh, or when you read a file with no extension that starts with a `#!` line and uses one of these shell names. In Shell mode, Epsilon does appropriate syntax highlighting, recognizing comments, variables and strings.

In Shell mode, Epsilon uses a tab size setting specified by the `shell-tab-override` variable.

When the cursor is on a brace, bracket, or parenthesis, Epsilon will try to locate its matching brace, bracket, or parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-shell-delimiters` to zero to disable this feature.

Summary: shell-mode

4.4.14   TeX and LaTeX Modes

Epsilon automatically enters either TeX or LaTeX mode when you read a file with an extension of .tex, .ltx, .sty, or (in most cases) .cls. (By default it enters LaTeX mode, but see the `tex-force-latex` command below.) TeX and LaTeX modes are almost identical, and will be described together.

Keys in TeX/LaTeX mode include Alt-i for italic text, Alt-Shift-I for slanted text, Alt-Shift-T for typewriter, Alt-Shift-B for boldface, Alt-Shift-C for small caps, Alt-Shift-F for a footnote, and Alt-s for a centered line.

Alt-Shift-E prompts for the name of a LaTeX environment, then inserts \begin{*env*} and \end{*env*} lines for the one you select. You can press ? to select an environment from a list. (The list of environments comes from the file `latex.env`, which you can edit.) Alt-Shift-Z searches backwards for the last \begin{*env*} directive without a matching \end{*env*} directive. Then it inserts the correct \end{*env*} directive at point.

For most of these commands, you can highlight a block of text first and Epsilon will insert formatting commands to make the text italic, slanted, etc. or you can use the command and then type the text to be italic, slanted, etc.

By default, Epsilon inserts the appropriate LaTeX 2e/3 command (such as \textit for italic text). Set the variable `latex-2e-or-3` to `0` if you want Epsilon to use the LaTeX 2.09 equivalent. (In the case of italic text, this would be \it.)

The keys '{' and '$' insert matched pairs of characters (either {} or $$). When you type \(or \[, TeX/LaTeX mode will insert a matching \) or \], respectively. But if you type '{' just before a non-whitespace character, it inserts only a '{'. This makes it easier to surround existing text with braces.

The keys ⟨Comma⟩ and ⟨Period⟩ remove a preceding italic correction \/, the " key inserts the appropriate kind of doublequote sequence like ' ' or ' ', and Alt-" inserts an actual " character.

Some TeX mode commands are slightly different in LaTeX than in pure TeX. Set `tex-force-latex` to `1` if all your documents are LaTeX, `0` if all your documents are TeX, or `2` if Epsilon should determine this on a document-by-document basis. In that case, Epsilon will assume a document is LaTeX if it contains a \begin{document} statement or if it's in a file with an .ltx, .sty, or .cls extension. By default, Epsilon assumes all documents use LaTeX.

When the cursor is on a curly brace or square bracket character like {, }, [, or], Epsilon will try to locate its matching character and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-tex-delimiters` to zero to disable this feature.

Set the variable `tex-look-back` to a bigger number if you want TeX mode to more accurately syntax highlight very large paragraphs but be slower, or a smaller number if you want recoloring to be faster but perhaps miscolor large paragraphs.

The compile-buffer command uses the `compile-tex-cmd` variable in TeX mode and the `compile-latex-cmd` variable in LaTeX mode. You may need to set these if the version of TeX or LaTeX you use takes some different flags. The MiKTeX version of TeX and LaTeX for Windows, for instance, works well with Epsilon if you use the flags "`-c-style-errors -interaction=nonstopmode`".

If your TeX system uses a compatible DVI previewer, then you can use Epsilon's jump-to-dvi command to see the DVI output resulting from the current line of TeX or LaTeX. This requires some setup so that the DVI file contains TeX source file line number data. See the jump-to-dvi command for details. With such setup, you can also configure your DVI viewer to run Epsilon, showing the source file and line corresponding to a certain spot in your DVI file. The details depend on your DVI viewer, but a command line like `epsilon -add +%l %f` is typical.

Summary: Alt-i   tex-italic

Alt-Shift-I	tex-slant
Alt-Shift-T	tex-typewriter
Alt-Shift-B	tex-boldface
Alt-Shift-C	tex-small-caps
Alt-Shift-F	tex-footnote
Alt-s	tex-center-line
Alt-Shift-E	tex-environment
Alt-Shift-Z	tex-close-environment
{	tex-left-brace
$	tex-math-escape
⟨Comma⟩, ⟨Period⟩	tex-rm-correction
"	tex-quote
Alt-"	tex-force-quote
\(	tex-inline-math
\[	tex-display-math
	tex-mode
	latex-mode
Alt-Shift-J	jump-to-dvi

4.4.15   VHDL Mode

Epsilon automatically enters VHDL mode when you read a file with an extension of .vhdl or .vhd. In VHDL mode, Epsilon does appropriate syntax highlighting. It also breaks and fills comment lines.

When the cursor is on a parenthesis, Epsilon will try to locate its matching parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-vhdl-delimiters` to zero to disable this feature.

Summary: vhdl-mode

4.4.16   Visual Basic Mode

Epsilon automatically enters Visual Basic mode when you read a file with an extension of .vb, .bas, .frm, .vbs, .aspx, .ctl, or .dsr (plus certain .cls files as well). In Visual Basic mode, Epsilon does appropriate syntax highlighting, smart indenting, tagging, and comment filling.

When the cursor is on a brace, bracket, or parenthesis, Epsilon will try to locate its matching brace, bracket, or parenthesis, and highlight them both. If the current character has no match, Epsilon will not highlight it. Set the variable `auto-show-vbasic-delimiters` to zero to disable this feature.

Set the `vbasic-indent` variable to alter the level of indentation Epsilon uses. Set `vbasic-indent-subroutines` to change Epsilon's indenting style. Set the `vbasic-indent-with-tabs` variable nonzero if you want Epsilon to indent using a mix of tab characters and spaces, instead of just spaces.

When you yank blocks of text into a buffer in Visual Basic mode, Epsilon can automatically reindent it. See the variable `reindent-after-vbasic-yank` to enable this behavior.

Summary: vbasic-mode

4.5   More Programming Features

Epsilon has a number of features that are useful when programming, but work similarly regardless of the programming language. These are described in the following sections. Also see the language-specific commands described in previous sections, and the tagging commands on page 43.

4.5.1   Navigating in Source Code

In most language modes, you can press Alt-' to display a list of all functions and global variables defined in the current file. You can move to a definition in the list and press ⟨Enter⟩ and Epsilon will go to that definition, or press Ctrl-G to remain at the starting point.

By default, this command skips over external declarations. With a prefix numeric argument, it includes those too. The `list-which-definitions` variable lets you customize which types of definitions are shown (in those modes that can show more than one type). The `list-definitions-live-update` variable lets you keep Epsilon from repositioning in the source file window as you navigate in the definitions window or include source file line numbers in the window title.

Also see Epsilon's tagging features on page 43.

Summary: Alt-'   list-definitions

4.5.2   Pulling Words

The pull-word command bound to the Ctrl-⟨Up⟩ key (as well as the F3 key) scans the buffer before point, and copies the previous word to the location at point. If you type the key again, it pulls in the word before that, and so forth. Whenever Epsilon pulls in a word, it replaces any previously pulled-in word. If you like the word that has been pulled in, you do not need to do anything special to accept it–Epsilon resumes normal editing when you type any key except for the few special keys reserved by this command. You can type Ctrl-⟨Down⟩ (the pull-word-fwd command) to go in the other direction. Type Ctrl-G to erase the pulled-in word and abort this command.

If a portion of a word immediately precedes point, that subword becomes a filter for pulled-in words. For example, suppose you start to type a word that begins `WM`, then you notice that the word `WM_QUERYENDSESSION` appears a few lines above. Just type Ctrl-⟨Up⟩ and Epsilon fills in the rest of this word.

The command provides various visual clues that tell you exactly from which point in the buffer Epsilon is pulling in the word. If the source is close enough to be visible in the window, it is simply highlighted. If the pulled-in word comes from farther away, Epsilon shows the context in the echo area, or in a context window that it pops up (out of the way of your typing).

When there are no more matches before point in the current buffer, Epsilon loads a tag file (see page 43) and looks for matches there. The `pull-word-from-tags` variable controls this behavior. In this way, you can complete on any tagged identifier by typing part of it and pressing F3 or Ctrl-⟨Up⟩ until the identifier you want appears.

You can also pull words from the buffer at most prompts. For instance, you can retrieve a long file name that appears in the buffer into a find-file prompt, or look for other instances of an identifier in a program

without typing the whole identifier. Type the first few characters at a search or grep prompt and press F3 or Ctrl-⟨Up⟩ to pull the rest.

Summary: Ctrl-⟨Up⟩, F3   pull-word
             Ctrl-⟨Down⟩   pull-word-fwd

4.5.3 Accessing Help

This section describes how Epsilon can help you access compiler help files and similar external documentation. See page 31 for directions on obtaining help on Epsilon itself.

Epsilon for Unix provides a man command for reading man pages. At its prompt, type anything you would normally type to the man command, such as -k open to get a list of man pages related to the keyword "open". If you don't use any flags or section names, Epsilon will provide completion on available topics. For example, type "?" to see all man page topics available. Within man page output, you can double-click on a reference to another man page, such as echo(1), or press ⟨Enter⟩ to follow it, or press m to be prompted for another man page topic. The man command also works with the Cygwin environment under Windows, if its man program is installed.

The search-man-pages command generates a list of man pages that contain some specified text, putting its results in the grep buffer. (See page 41.) It first prompts for the search string. You can use Ctrl-T, Ctrl-C, or Ctrl-W to toggle regular expression, case folding, or word searching behavior, as with grep and other searching commands.

Then it asks if you want to restrict searching to particular man page sections, such as 1 for commands or 3 for subroutines. Use * to search all sections. Finally, it asks if you want to restrict the search to man page entries matching a certain file pattern, such as *file* to search only pages whose names contain "file".

For speed reasons, it searches each man page without processing it through the man command, searching the man page in source format. By default, it shows only the first match in each page; set the search-man-pages-shows-all variable to see all matches. The result appears in the grep buffer; when you view a match from there, Epsilon will then use the man command to display its processed form.

Epsilon also includes a perldoc command for reading Perl documentation. Just like man, it works by running an external program, in this case the perldoc program that comes with Perl. Run perldoc on the topic perldoc to see the flags you can use with it, such as -f to locate the documentation for a specific Perl function.

You can set up Epsilon for Windows to search for help on a programming language construct (like an API function or a C++ keyword) in a series of help files. Epsilon can link to both .hlp and .chm (HtmlHelp) files. Run the Select Help Files... command on the help menu to select the help files you want to use. This command adds help files to the Help menu, to the context menu that the secondary mouse button displays, and to the list of files searched by the Search All Help Files... command on the help menu. The last command is only available under Windows. Edit the file gui.mnu to further modify the contents of Epsilon's menus. Edit the file epswhlp.cnt to modify the list of files searched by Search All Help Files.

If you highlight a word in the buffer before running a help command, Epsilon will search for help on that keyword. Otherwise Epsilon will display either a list of available keywords or the table of contents for the help file you selected.

Summary: man
                                              search-man-pages
                                              select-help-files

search-all-help-files

4.5.4   Commenting Commands

The Alt-; command creates a comment on the current line, using the commenting style of the current language mode. The comment begins at the column specified by the `comment-column` variable (by default 40). (However, if the comment is the first thing on the line and `indent-comment-as-code` is nonzero, it indents to the column specified by the buffer's language-specific indentation function.) If the line already has a comment, this command moves the comment to the comment column.

With a numeric argument, Alt-; searches for the next comment in the buffer and goes to its start. With a negative argument, Alt-; searches backwards for a comment. Press Alt-; again to reindent the comment.

By default (and in modes that don't specify a commenting style), comments begin with the ; character and continue to the end of the line. C mode recognizes both old-style /* */ comments, and the newer C++-style comments //, and by default creates the latter. Set the variable `new-c-comments` to 0 if you want Alt-; to create old-style comments.

The Ctrl-X ; command sets future comments to begin at the current column. With a positive argument, it sets the comment column based on the indentation of the previous comment in the buffer. If the current line has a comment, this command reindents it.

With a negative argument (as in Alt-⟨Minus⟩ Ctrl-X ;), the Ctrl-X ; command doesn't change the comment column at all. Instead, it kills any comment on the current line. The command saves the comment in a kill buffer.

You can comment out a region of text by pressing Ctrl-C Ctrl-R. Epsilon adds a comment delimiter to the start of each line in the region between point and mark. With a numeric argument, as in Ctrl-U Ctrl-C Ctrl-R, Epsilon removes such a comment delimiter from each line.

The comment commands look for comments using regular expression patterns (see page 55) contained in the buffer-specific variables `comment-pattern` (which should match the whole comment) and `comment-start` (which should match the sequence that begins a comment, like '/*'). When creating a comment, it inserts the contents of the buffer-specific variables `comment-begin` and `comment-end` around the new comment. When Epsilon puts a buffer in C mode, it decides how to set these variables based on the `new-c-comments` variable.

In certain modes, including C and Perl modes, Epsilon normally auto-fills text in block comments as you type, breaking overly long lines. See the `c-auto-fill-mode` variable for C and Perl, `tex-auto-fill-mode` for TeX, `html-auto-fill-mode` for HTML and XML, and `misc-language-fill-mode` in Makefile, VHDL, Visual Basic, Python, PostScript, and Conf modes. As with normal auto-fill mode (see page 65), use Ctrl-X F to set the right margin for filling. Set the `c-fill-column` variable to change the default right margin in C and Perl mode buffers; `margin-right` in other modes.

You can manually refill the current paragraph in a block comment by pressing Alt-q. If you provide a numeric prefix argument to Alt-q, say by typing Alt-2 Alt-q, it will fill using the current column as the right margin. By default, Epsilon doesn't apply auto-filling to a comment line that also contains non-comment text (such as a C statement with a comment after it on the same line). Use Alt-q to break such lines.

The `auto-fill-comment-rules` variable lets you customize certain aspects of Epsilon's behavior when breaking and filling comment lines.

Summary: Alt-;   indent-for-comment
                  Ctrl-X ;   set-comment-column

| Alt-⟨Minus⟩ Ctrl-X ; | kill-comment |
| Ctrl-C Ctrl-R | comment-region |

4.6 Fixing Mistakes

4.6.1 Undoing

The undo command on F9 undoes the last command, restoring the previous contents of the buffer, or moving point to its position, as if you hadn't done the last command. If you press F9 again, Epsilon will undo the command before that, and so forth.

For convenience, when typing text Epsilon treats each word you type as a single command, rather than treating each character as its own command. For example, if you typed the previous paragraph and pressed undo, Epsilon would remove the text "forth.". If you pressed undo again, Epsilon would remove "so ".

Epsilon's undo mechanism considers each subcommand of a complicated command such as query-replace a separate command. For example, suppose you do a query-replace, and one-by-one replace ten occurrences of a string. The undo command would then reverse the replacements one at a time.

Epsilon remembers changes to each buffer separately. Say you changed buffer 1, then changed buffer 2, then returned to buffer 1. Undoing now would undo the last change you made to buffer 1, leaving buffer 2 alone. If you switched to buffer 2 and invoked undo, Epsilon would then undo changes to that buffer.

The redo command on F10 puts your changes back in (it undoes the last undo). If you press undo five times, then press redo four times, the buffer would appear the same as if you pressed undo only once.

You can move back and forth undoing and redoing in this way. However, if you invoke a command (other than undo or redo) that either changes the buffer or moves point, you can not redo any commands undone immediately before that command. For example, if you type "one two three", undo the "three", and type "four" instead, Epsilon will behave as if you had typed "one two four" all along, and will let you undo only that.

The commands undo-changes and redo-changes work like undo and redo, except they will automatically undo or redo all changes to the buffer that involve only movements of point, and stop just before a change of actual buffer contents.

For example, when you invoke undo-changes, it performs an undo, then continues to undo changes that involve only movements of point. The undo-changes command will either undo a single buffer modification (as opposed to movement of point), as a plain undo command would, or a whole series of movement commands at once. It doesn't undo any movement commands after undoing a buffer modification, only after undoing other movement commands. The redo-changes command works similarly.

The Ctrl-F9 key runs undo-changes, and the Ctrl-F10 key runs redo-changes.

The commands undo-by-commands and redo-by-commands are another alternative; they try to group undo operations on a command-by-command basis.

The buffer-specific variable undo-size determines, in part, how many commands Epsilon can remember. For example, if undo-size has the value 500,000 (the default), Epsilon will save at most 500,000 characters of deleted or changed text for each buffer. Each buffer may have its own value for this variable. Epsilon also places an internal limit on the number of commands, related to command complexity. Epsilon can typically remember about 10,000 simple commands (ignoring any limit imposed by undo-size) but more complicated commands make the number smaller.

Summary: F9, Ctrl-X U   undo

F10, Ctrl-X R	redo
Ctrl-F9, Ctrl-X Ctrl-U	undo-changes
Ctrl-F10, Ctrl-X Ctrl-R	redo-changes
	undo-by-commands
	redo-by-commands

4.6.2   Interrupting a Command

You can interrupt a command by pressing Ctrl-G, the default *abort key*. For example, you can use Ctrl-G to stop an incremental search on a very long file if you don't feel like waiting. You can set the abort key with the set-abort-key command. If you interrupt Epsilon while reading a file from disk or writing a file to disk, it will ask you whether you want to abort or continue. Typing the abort key also cancels any currently executing keyboard macros. Aborting normally only works when a command checks for it.

Summary:	Ctrl-G	abort
		set-abort-key

4.7   The Screen

4.7.1   Display Commands

The Ctrl-L command causes Epsilon to center point in the window. If you give a numeric argument to Ctrl-L, Epsilon makes the current line appear on that line of the window. For instance, give a numeric argument of zero to make the current line appear on the topmost line of the window. (The line-to-top command is another way to do this.) If you give a numeric argument greater than the number of lines the window occupies, Epsilon will position the current line at the bottom of the window. (The line-to-bottom command is another way to do this.) When repeated, the Ctrl-L command also completely refreshes the screen. If some other program has written text on the screen, or something has happened to garble the screen, use this command to refresh it.

The Alt-⟨Comma⟩ and Alt-⟨Period⟩ commands move point to the first and last positions displayed on the window, respectively.

The Ctrl-Z and Alt-Z commands scroll the text in the window up or down, respectively, by one line. These scrolling commands will move point as necessary so that point remains visible in the window.

The Ctrl-V and Alt-V commands scroll the text of the window up or down, respectively, by several lines fewer than the size of the window. These commands move point to the center line of the window.

You can control the exact amount of overlap between the original window of text and the new window with the `window-overlap` variable. A positive value for this variable means to use that number of screen lines of overlap between one window of text and the next (or previous). A negative value for `window-overlap` represents a percentage of overlap, instead of the number of screen lines. For example, the default value for `window-overlap` of 2 means to use 2 lines of overlap. A value of -25 for `window-overlap` means to overlap by 25%.

You can change how Epsilon pages through a file by setting the variable `paging-centers-window`. Epsilon normally positions the cursor on the center line of the window as you move from page to page. Set this variable to zero if you want Epsilon to try to keep the cursor on the same screen line as it pages.

The goto-line command on Ctrl-X G prompts for a line number and then goes to the beginning of that line in the current buffer. If you prefix a numeric argument, Epsilon will use that as the line number. Use the format 10:20 to include a column specification; that one goes to line 10, column number 20. Or use a percent character to indicate a buffer percentage: 25% goes to a line 25% of the way through the buffer. Or use the format p123 to go to a particular buffer offset, counting by characters.

The Ctrl-X L command shows the number of lines in the buffer and the number of the line containing point. It also shows the number of bytes the file would occupy if written to disk. This can differ from the size of the buffer, because the latter counts each line separator as a single character. Such characters require two bytes when written to disk in the format used in Windows, DOS, and OS/2, however. See page 100 for information on how Epsilon translates line separator characters. See mode-format if you want Epsilon to display the current line or column number at all times.

The Ctrl-X = command displays in the echo area information pertaining to point. It shows the size of the buffer, the character position in the buffer corresponding to point, that character's column, and the value of that character in decimal, hex, and "normal" character representation.

Summary:		
	Ctrl-L	center-window
	Ctrl-V, ⟨PgDn⟩	next-page
	Alt-V, ⟨PgUp⟩	previous-page
	Ctrl-Z	scroll-up
	Alt-Z	scroll-down
	⟨Home⟩, Alt-⟨Comma⟩	beginning-of-window
	⟨End⟩, Alt-⟨Period⟩	end-of-window
		line-to-top
		line-to-bottom
	Ctrl-X =	show-point
	Ctrl-X L	count-lines
	Ctrl-X G	goto-line

4.7.2 Horizontal Scrolling

The Alt-{ and Alt-} commands scroll the text in the window to the left or right, respectively, by one column.

The Alt-{ and Alt-} commands also control how Epsilon displays long lines to you. Epsilon can, for display purposes, wrap long lines to the next line. Epsilon indicates a wrapped line by displaying a special continuation character where it broke the line for display purposes. But by default Epsilon displays long lines by simply scrolling them off the display. To switch from scrolling long lines to wrapping long lines, use the Alt-} command to scroll to the right, past the end. Epsilon will then wrap long lines.

Similarly, to switch from wrapping long lines to scrolling long lines, press the Alt-{ key. Subsequent use of the Alt-{ command will then scroll the text in the window to the left, as explained above. Whenever Epsilon changes from one display scheme to the other, it indicates the change in the echo area. If, due to scrolling, some of a buffer's contents would appear past the left edge of the screen, the mode line displays "<*number*" to indicate the number of columns hidden to the left.

You can also use the change-line-wrapping command to set whether Epsilon wraps long lines in the current window, or horizontally scrolls across them.

If you want Epsilon to always wrap long lines, set the default value of the window-specific variable display-column to −1 using the set-variable command on F8, then save the state using the write-state command on Ctrl-F3.

In a dialog, another way to handle lines that are too long to fit in a window is to resize the dialog by moving its borders. Most dialogs in Epsilon for Windows are resizable, and Epsilon will remember the new size from session to session.

The Alt-PageUp and Alt-PageDown keys scroll horizontally, like Ctrl-V and Alt-V. More precisely, they move the point left or right on the current line by about half the width of the current window, then reposition the window so the point is visible. The command jump-to-column on Alt-g prompts for a column number, then goes to the specified column.

Summary:	Alt-{	scroll-left
	Alt-}	scroll-right
		change-line-wrapping
	Alt-⟨PageUp⟩	page-left
	Alt-⟨PageDown⟩	page-right
	Alt-g	jump-to-column

4.7.3   Windows

Epsilon has quite a few commands to deal with creating, changing, and moving windows. Changing the size or number of the windows never affects the buffers they display.

Normally, each buffer has a single point, but this can prove inconvenient when a buffer appears in more than one window. For this reason, Epsilon associates a point with each window in that case. Consequently, you can look at different parts of the same buffer by having the same buffer displayed in different windows and moving around independently in each of them.

Creating Windows

The Ctrl-X 2 command splits the current window into two windows, one on top of the other, each about half as large. Each window displays the same buffer that the original did. This command will only split the window if each new window would occupy at least 1 screen line, not counting the mode line. To edit another file in a new window, first use Ctrl-X 2, then use one of the file commands described on page 95.

The Ctrl-X 5 command works similarly, but splits the current window so that the two child windows appear side by side, instead of stacked. This command will only split the window if each new window would occupy at least 1 column. Since this typically results in narrow windows, the Ctrl-X 5 command also sets up the windows to scroll long lines, as described on page 85.

Summary:	Ctrl-X 2	split-window
	Ctrl-X 5	split-window-vertically

Removing Windows

To get rid of the current window, use the Ctrl-X 0 command. If the previous window can move into the deleted window's space, it does. Otherwise, the next window expands into the deleted window's space.

The Ctrl-X 1 command makes the current window occupy the entire screen, deleting all the other windows. The Ctrl-X Z command operates like Ctrl-X 1, except that it also remembers the current window configuration. Later, if you type Ctrl-X Z again, the command restores the saved window configuration.

Summary: Ctrl-X 0, Ctrl-X Ctrl-D   kill-window
                         Ctrl-X 1   one-window
                         Ctrl-X Z   zoom-window

Selecting Windows

The Ctrl-X N key moves to the next window, wrapping around to the first window if invoked from the last window. The Ctrl-X P key does the reverse: it moves to the previous window, wrapping around to the last window if invoked from the first window.

You can think of the window order as the position of a window in a list of windows. Initially only one window appears in the list. When you split a window, the two child windows replace it in the list. The top or left window comes before the bottom or right window. When you delete a window, that window leaves the list.

You can also change windows with the move-to-window command. It takes a cue from the last key in the sequence used to invoke it, and moves to a window in the direction indicated by the key. If you invoke the command with Ctrl-X ⟨Right⟩, for example, the window to the right of the cursor becomes the new current window. The Ctrl-X ⟨Left⟩ key moves left, Ctrl-X ⟨Up⟩ moves up, and Ctrl-X ⟨Down⟩ moves down. If key doesn't correspond to a direction, the command asks for a direction key.

Summary: Alt-⟨End⟩, Ctrl-X N   next-window
                         Alt-⟨Home⟩, Ctrl-X P   previous-window
                         Ctrl-X ⟨Up⟩, Ctrl-X ⟨Down⟩   move-to-window
                         Ctrl-X ⟨Left⟩, Ctrl-X ⟨Right⟩   move-to-window

Resizing Windows

The easiest way to resize Epsilon windows is to use the mouse. But Epsilon also provides various ways to do this via the keyboard.

The Ctrl-X + key runs the command enlarge-window-interactively. After you invoke the command, point to a window border using the arrow keys. The indicated window border moves so as to make the current window larger. You can keep pressing arrow keys to enlarge the window. To switch from enlarging to shrinking, press the minus key. The command Ctrl-X – works like Ctrl-X +, but starts out shrinking instead of enlarging. Whenever the window looks the right size, press ⟨Enter⟩ to leave the command.

You can use several other Epsilon commands to resize windows. The Ctrl-⟨PgUp⟩ key enlarges the current window vertically, and the Ctrl-⟨PgDn⟩ key shrinks the current window vertically. They do this by moving the mode line of the window above them up or down, if possible. Otherwise, the current window's mode line moves up or down, as appropriate.

You can also enlarge and shrink windows horizontally. The enlarge-window-horizontally command on Ctrl-X @ enlarges the current window by one column horizontally and the shrink-window-horizontally command shrinks it. They do this by moving the left boundary of the current window left or right, if possible. Otherwise, the current window's right boundary moves, as appropriate. You can use a numeric prefix with these commands to adjust by more than one line or column, or in the opposite direction.

Summary: Ctrl-X +   enlarge-window-interactively
                         Ctrl-X –   shrink-window-interactively

Ctrl-⟨PgUp⟩, Ctrl-X ^	enlarge-window
Ctrl-⟨PgDn⟩	shrink-window
Ctrl-X @	enlarge-window-horizontally
	shrink-window-horizontally

4.7.4   Customizing the Screen

Epsilon displays tabs in a file by moving over to the next tab stop column. Epsilon normally spaces tabs every four or eight columns, depending on the mode. You can change the tab stop spacing by setting the variable `tab-size`. Another method is to use the set-tab-size command, but this can only set the tab size in the current buffer. To change the default value for new buffers, set the variable using the set-variable command.

Many indenting commands take the tab size into account when they indent using spaces and tabs. See page 67 for information on the indenting commands.

Epsilon can display special characters in four ways. Epsilon normally displays control characters with a ^ prefix indicating a control character (except for the few control characters like ^I that have a special meaning—^I, for example, means ⟨Tab⟩). It displays other characters, including national characters, with their graphic symbol.

In mode 0, Epsilon displays Meta characters (characters with the 8th bit on) by prefixing to them a "M-", e.g., Meta C appears as "M-C". Epsilon display Control-meta characters by prefixing to them "M-^", e.g., "M-^C". Epsilon displays most control characters by prefixing to them a caret, e.g., Control C appears as "^C".

In mode 1, Epsilon displays graphic symbols for all control characters and meta characters, instead of using a prefix as in ^A (except for the few that have a special meaning, like ⟨Tab⟩ or ⟨Newline⟩).

In mode 2, Epsilon displays control and meta characters by their hexadecimal ASCII values, with an "x" before them to indicate hex.

In mode 3, which is the default, Epsilon displays control characters as "^C", and uses the graphic symbol for other characters, as described above.

The set-show-graphic command on Ctrl-F6 cycles among these four modes of representation. Providing a numeric argument of 0, 1, 2, or 3 selects the corresponding mode.

The command change-show-spaces on Shift-F6 makes spaces, tabs, and newline characters in the buffer visible, by using special graphic characters for each. Pressing it again makes these characters invisible. The command sets the buffer-specific variable `show-spaces`.

Epsilon will usually display a message in the echo area for at least one second before replacing it with a new message. You can set this time with the `see-delay` variable. It contains the number of hundredths of a second that a message must remain visible, before a subsequent message can overwrite it. Whenever you press a key with messages pending, Epsilon skips right to the last message and puts that up. (Epsilon doesn't stop working just because it can't put up a message; it just remembers to put the message up later.)

You can set variables to modify the text cursor shape Epsilon displays in different situations. Epsilon gets the cursor shape from one of four variables, depending upon whether or not Epsilon is in overwrite mode, and whether or not the cursor is positioned in virtual space. (See the description of the `virtual-space` variable on page 36.) These variables only apply in text mode, not in Epsilon for Windows or under X in Unix, and in some environments have no effect.

Variable	In overwrite mode?	In virtual space?
`normal-cursor`	No	No
`overwrite-cursor`	Yes	No
`virtual-insert-cursor`	No	Yes
`virtual-overwrite-cursor`	Yes	Yes

Each of these variables contains a code that specifies the top and bottom edges of the cursor, such as 3006, which specifies a cursor that begins on scan line 3 and extends to scan line 6 on a character box. The topmost scan line is scan line 0.

Scan lines above 50 in a cursor shape code are interpreted differently. A scan line number of 99 indicates the highest-numbered valid scan line (just below the character), 98 indicates the line above that, and so forth. For example, a cursor shape like 1098 produces a cursor that extends from scan line 1 to the next-to-last scan line, one scan line smaller at top and bottom than a full block cursor.

The Windows and X versions of Epsilon use a similar set of variables to control the shape of the cursor (or caret, in Windows terminology).

Variable	In overwrite mode?	In virtual space?
`normal-gui-cursor`	No	No
`overwrite-gui-cursor`	Yes	No
`virtual-insert-gui-cursor`	No	Yes
`virtual-overwrite-gui-cursor`	Yes	Yes

Each variable contains a code that specifies the height and width of the caret, as well as a vertical offset, each expressed as a percentage of the character dimensions. Values close to 0 or 100 are absolute pixel counts, so a width of 98 is two pixels smaller than a character. A width of exactly zero means use the default width.

All measurements are from the top left corner of the character. A nonzero vertical offset moves the caret down from its usual starting point at the top left corner.

In EEL programs, you can use the `GUI_CURSOR_SHAPE()` macro to combine the three values into the appropriate code; it simply multiplies the height by 1000 and the offset by 1,000,000, and adds both to the width. So the default Windows caret shape of `GUI_CURSOR_SHAPE(100, 2, 0)`, which specifies a height of 100% of the character size and a width of 2 pixels, is encoded as the value 100,002. The value 100100 provides a block cursor, while 99,002,100 makes a good underline cursor. (It specifies a width of 100%, a height of 2 pixels, and an offset of 99 putting the caret down near the bottom of the character cell.) The `CURSOR_SHAPE()` macro serves a similar purpose for text mode versions of Epsilon.

Epsilon for Windows can draw a rectangle around the current line to increase its visibility and make it easier to find the cursor. Set the `draw-focus-rectangle` variable nonzero to enable this. Set the `draw-column-markers` variable if you want Epsilon for Windows to draw a vertical line at a particular column or columns, to make it easier to edit text that must be restricted to certain columns. (Also see auto-fill mode described on page 65.)

The set-display-characters command lets you alter the various characters that Epsilon uses to construct its display. These include the line-drawing characters that form window borders, the characters Epsilon uses in some of the display modes set by set-show-graphic, the characters it uses to construct the scroll bar, and the characters Epsilon replaces for the graphical mouse cursor it normally uses in DOS. The command displays a matrix of possible characters, and guides you through the selection process.

Summary:　　　　Ctrl-F6　　　　　　　　　　　　set-show-graphic

Shift-F6	change-show-spaces
	set-tab-size
	set-display-characters

4.7.5 Fonts

The set-font command changes the font Epsilon uses, by displaying a font dialog box and letting you pick a new font. Modifying the `font-fixed` variable is another way to set the font.

Epsilon also applies the styles bold, italic and underlined to the selected font for certain types of text, such as comments. Epsilon treats these styles as if they were part of the foreground color of a particular type of text. The set-color command lets you set which types of text receive which styles. Also see the variables `font-styles` and `font-styles-tolerance`.

Epsilon for Unix supports setting the font under X, using set-font or `font-fixed`, but not the remaining commands and settings in this section.

You can specify a specific font for use in printing with the set-printer-font command. Similarly, the set-dialog-font command lets you specify what font to use for Epsilon's dialog windows (like the one bufed displays). There are also corresponding variables `font-printer` and `font-dialog`.

The command change-font-size supplements set-font by providing additional font choices. Some Windows fonts include a variety of character cell widths for a given character cell height. (For example, many of the font selections available in windowed DOS sessions use multiple widths.) Commands like set-font utilize the standard Windows font dialog, which doesn't provide any way to select these alternate widths. The change-font-size command lets you choose these fonts.

The change-font-size command doesn't change the font name, or toggle bold or italic. You'll need to use the set-font command to do that.

Instead, change-font-size lets you adjust the height and width of the current font using the arrow keys. You can abort to restore the old font settings, or press ⟨Enter⟩ or ⟨Space⟩ to keep them. This is a handy way to shrink or expand the font size. A width or height of 0 means use a suitable default.

Summary:	set-font
	set-printer-font
	set-dialog-font
	change-font-size

4.7.6 Setting Colors

This section describes how to set colors in Epsilon. Epsilon comes with many built-in color schemes. Each *color scheme* tells Epsilon what color to use for each *color class*. Color classes correspond to the different parts of the screen. There are separate color classes for normal text, highlighted text, text in the echo area, syntax-highlighted comments, and so forth. (See below for a partial list.)

You can select a different color scheme using the set-color command. In text-based versions of Epsilon that don't display a color-setting dialog window, use the F and B keys to move forward and backward in the list of color schemes, or select a new one with the mouse.

In some environments, you can also select font styles bold, italic and underlined for each color class. If the current font doesn't provide font styles that are sized close enough to the base font, some styles will not be available. See the `font-styles-tolerance` variable.

Epsilon remembers the name of one color scheme for use on text mode displays with only 8 or 16 possible color choices, and a separate scheme for environments like Windows or X where it can display all possible colors. (It also maintains separate schemes for monochrome displays, and for when Epsilon runs as a Unix terminal program within an xterm and the USE_DEFAULT_COLORS environment variable is defined; the latter enables a special color scheme that's designed to inherit the background and foreground colors of the underlying xterm.)

When you've turned off window borders with the toggle-borders command, Epsilon uses color schemes with particular, fixed names. See page 93.

Use the set-color command to select a color scheme from the list of available color schemes. You can also customize a color scheme by selecting one, selecting a color class within it, using the buttons to select a different foreground or background color, and toggling whether that color class tries to use bold, italic, or underlined variations of the current font.

The Unix and Win32 console versions of set-color uses a slightly different user interface than the Windows GUI and X versions. In the console versions, you can select the color scheme in the Color Scheme window with the F and B keys. Then select the particular color class you want to modify by pressing the N and P keys. Finally, use the arrow keys to move about in the matrix of color combinations that Epsilon displays. You can also select a color scheme, color class, or color combination with the mouse.

Another method of customizing a color scheme is to create an EEL file like stdcolor.e. The file stdcolor.e defines all Epsilon's built-in color schemes. You can use one of these as a model for your own color scheme. See page 183 for the syntax of color scheme definitions. You can use the export-colors command to build an EEL file named mycolors.e that contains all Epsilon's current color definitions for the current color scheme. (With a numeric argument, it lists all schemes.)

The Win32 console and Unix terminal versions of Epsilon are limited to the sixteen standard colors for foreground and background, for a total of 256 possible color combinations, while Epsilon for Windows (and Epsilon for Unix, as an X program) have no such limitation. Internally, all versions of Epsilon store 32 bits of color information for the foreground and background of each color class. The console and terminal versions convert back to 4 bits of foreground and background when displaying text.

In some environments, Epsilon provides eight possible background colors; in others it provides sixteen, for a total of 256 possible foreground/background combinations. The variable selectable-colors controls the number of colors the set-color command lets you select from. Epsilon sets it to 256 instead of 128 on appropriate systems. The command still only displays 128 combinations at a time. The ⟨Up⟩ and ⟨Down⟩ keys flip to the other 128 possibilities, or use the mouse to scroll the color window.

The set-color command displays a short description of each color class as you select it. Here we describe a few of the color classes in more detail:

text Epsilon puts the text of an ordinary buffer in this color. But if Epsilon is doing code coloring in a buffer, it uses the color classes defined for code coloring instead. For C, C++, Java, and EEL files, these all start with "c-" and appear farther down in the list of color classes.

mode-line Epsilon uses this color for the text in the mode line of a tiled window.

horiz-border Epsilon uses this color for the line part of the mode line of a tiled window.

vert-border Epsilon uses this color for the vertical border it draws between tiled windows.

after-exiting Some console versions of Epsilon try to leave the screen in this color when you exit. Epsilon normally sets this color when it starts up, based on the screen's colors before you started Epsilon. Set the restore-color-on-exit variable to zero to disable this behavior, so you can set the color explicitly and preserve the change in your state file.

`debug-text` The EEL debugger uses this color when it displays EEL source code.

`default` Epsilon initializes any newly-defined color classes (see page 90) with this color.

`screen-border` Epsilon sets the border area around the screen or window to match this color's background. Epsilon only uses the background part of this color; the foreground part doesn't matter.

`screen-decoration` Epsilon for Windows can draw a focus rectangle or column markers. The foreground color specified here determines their color. See the `draw-focus-rectangle` and `draw-column-markers` variables.

`pull-highlight` The pull-word command uses this color for its highlighting.

Summary: set-color

                                                                                 export-colors

4.7.7 Code Coloring

Epsilon does syntax-based highlighting of C, C++, Java, and EEL files. Set the buffer-specific variable `want-code-coloring` to 0 to disable this feature or run the change-code-coloring command. To change the colors Epsilon uses, see the previous section.

If you use a slower computer, you may need to tell Epsilon to do less code coloring, in order to get acceptable response time. Set the variable `minimal-coloring` to 1 to tell Epsilon to look only for comments, preprocessor lines, strings, and character constants when coloring. Epsilon will color all identifiers, functions, keywords, numbers and punctuation the same, using the `c-ident` color class for all. This makes code coloring much faster.

When Epsilon begins coloring in the middle of a buffer, it has to determine whether it's inside a comment by searching back for comment characters. If you edit extremely large C files with few block comments, you can speed up Epsilon by telling it not to search so far. Set the variable `color-look-back` to the number of characters Epsilon should search through before giving up. Any block comments larger than this value may not be colored correctly. A value of zero (the default) lets Epsilon search as far as it needs to, and correctly colors comments of any size.

When Epsilon isn't busy acting on your keystrokes, it looks through the current buffer and assigns colors to the individual regions of text, so that Epsilon responds faster as you scroll through the buffer. For smoother performance, Epsilon doesn't begin to do this until it's been idle for a certain period of time, contained in the `idle-coloring-delay` variable. This holds the number of hundredths of a second to wait before computing more coloring information. By default, it's 100, so Epsilon waits one second. Set it to -1 to disable background code coloring.

Normally Epsilon colors buffers as needed. You can set Epsilon to instead color the entire buffer the first time it's displayed. Set the variable `color-whole-buffer` to the size of the largest buffer you want Epsilon to entirely color at once.

Summary: change-code-coloring

4.7.8 Window Borders

Use the command set-display-look to make Epsilon's window decoration and screen appearance resemble that of other editors. It displays a menu of choices. You can select Epsilon's original look, Brief's look, the look of the DOS Edit program (the same as the QBasic program), or the look of the Borland IDE.

The command toggle-borders removes the lines separating Epsilon's windows from one another, or restores them.

When there are no window borders, Epsilon provides each window with its own separate color scheme, in place of the single one selected by set-color. (You can still use set-color to set the individual colors in a color scheme, but Epsilon doesn't care which particular color scheme you select when it displays the contents of individual windows. It does use your selected color scheme for other parts of the screen like the echo area or screen border.)

The color schemes Epsilon uses for borderless windows have names like "`window-black`", "`window-blue`" and so forth. Epsilon assigns them to windows in order. You can remove one from consideration using the delete-name command, or create a new one using EEL (see page 183).

The rest of this section describes some of the variables set by the above commands. The set-display-look command in particular does its work entirely by setting variables. You can make Epsilon use a custom display look by setting these variables yourself. The variables also allow some customizations not available through the above commands.

The `echo-line` variable contains the number of the screen line on which to display the echo area. The `avoid-top-lines` and `avoid-bottom-lines` variables tell Epsilon how many screen lines at the top and bottom of the screen are reserved, and may not contain tiled windows. By default, `echo_line` contains the number of the last screen line, `avoid-top-lines` is zero, and `avoid-bottom-lines` is one, to make room for the echo area.

To Epsilon display text in the echo area whenever it's idle, set the variables `show-when-idle` and `show-when-idle-column`. See their online documentation for details.

To position the echo area at the top of the screen, set `echo-line` and `avoid-bottom-lines` to zero and `avoid-top-lines` to one. (If you're using a permanent mouse menu, set `echo-line` and `avoid-top-lines` one higher.)

To completely fill the screen with text, toggle borders off and set `avoid-bottom-lines` and `avoid-top-lines` to zero. Whenever Epsilon needs to display text in the echo area, it will temporarily overwrite the last screen line for a moment, and then return to showing buffer text on every line.

You can customize the position and contents of the mode line Epsilon displays for ordinary tiled windows by setting variables. These variables all start with "mode-". See the online help for `mode-format` for details. Also see the `full-path-on-mode-line` variable.

You can set several variables to put borders around the screen. If you want Epsilon to always display a window border at the right edge of the screen, set the variable `border-right` nonzero. (The toggle-scroll-bar command, which turns on permanent scroll bars for all windows, sets this variable.) Epsilon displays a border at the left screen edge if `border-left` has a nonzero value. Similarly, `border-top` and `border-bottom` variables control borders at the top and bottom edges of the screen, but only if a tiled window reaches all the way to that edge of the screen. (A menu bar might be in the way.) All these variables are zero by default. (Toggling all window borders off with the toggle-borders command overrides these variables.) If the `border-inside` variable is nonzero (as it is by default), Epsilon displays a border between side-by-side windows. Set it to zero to eliminate these borders. (The toggle-borders command sets this variable, among other things.)

Summary: set-display-look

4.7.9   The Bell

Sometimes Epsilon will ring the computer's bell to alert you to certain conditions. (Well, actually it sounds more like a beep, but we call it a bell anyway.) You can enable or disable the bell completely by setting the `want-bell` variable. Epsilon will never try to beep if `want-bell` has a value of zero.

For finer control of just when Epsilon rings the bell, you can set the variables listed in figure 4.6 using the set-variable command, described on page 127. A nonzero value means Epsilon will ring the bell when the indicated condition occurs. By default, all these variables but `bell-on-abort` have the value 1, so Epsilon rings the bell on almost all of these occasions.

Variable	When Epsilon Beeps, if Nonzero
`bell-on-abort`	You abort with Ctrl-G, or press an unbound key.
`bell-on-autosave-error`	Autosaving can't write files.
`bell-on-bad-key`	You press an illegal option at a prompt.
`bell-on-completion`	Completion finds no matches.
`bell-on-date-warning`	Epsilon notices that a file has changed on disk.
`bell-on-read-error`	Epsilon cannot read a file.
`bell-on-search`	Search finds no more matches.
`bell-on-write-error`	Epsilon cannot write a file.

Figure 4.6: Variables that control when Epsilon rings the bell

In some environments, the `beep-duration` variable specifies the duration of the beep, in hundredths of a second. The `beep-frequency` variable specifies the frequency of the bell in hertz.

Instead of making a sound for the bell, you can have Epsilon invert the mode line of each window for a time according to the value of `beep-duration` by setting `beep-frequency` to zero, and `beep-duration` to any nonzero value.

Under Windows, Epsilon doesn't use the `beep-duration` or `beep-frequency` variables. It uses a standard system sound instead. Under Unix, Epsilon recognizes a `beep-frequency` of zero and flashes the screen in some fashion, but otherwise ignores these variables.

4.8   Buffers and Files

4.8.1   Buffers

The Ctrl-X B command prompts you for a buffer name. The command creates a buffer if one with that name doesn't already exist, and connects the buffer to the current window.

The new-file command creates a new buffer and marks it so that Epsilon will prompt for its file name when you try to save it. It doesn't prompt for a buffer name, unlike Ctrl-X B, but chooses an unused name. Another difference is that Epsilon will warn about saving changes to a buffer created by new-file, while a buffer created by Ctrl-X B is treated as a scratch buffer.

You can customize the behavior of the new-file command by setting the variables `new-file-mode` and `new-file-ext`. The `new-file-mode` variable contains the name of the mode-setting command Epsilon should use to initialize new buffers; the default is the c-mode command. The `new-file-ext` variable contains the extension of the file name Epsilon constructs for the new buffer; its default is ".c".

To get a list of the buffers, type Ctrl-X Ctrl-B. This runs the bufed (for buffer edit) command, described fully on page 112. Basically, bufed lists your buffers, along with their sizes and the files (if any) contained in those buffers. You can then easily switch to any buffer by positioning point on the line describing the buffer and pressing the ⟨Space⟩ key. The bufed command initially positions point on the buffer from which you invoked bufed. Press Ctrl-G if you decide not to switch buffers after all.

The bufed command usually does not list special buffers such as the kill buffers. If you prefix a numeric argument, however, bufed shows all the buffers.

The Ctrl-X K command eliminates a buffer. It asks you for a buffer name and gets rid of it. If the buffer has unsaved changes, the command warns you first.

The Ctrl-X Ctrl-K command eliminates the current buffer, just like Ctrl-X K, but without asking which buffer you want to get rid of. The kill-all-buffers command discards all user buffers.

Whenever Epsilon asks you for a buffer name, it can do completion on buffer names, and will list matches in a pop-up window if you press '?'.

Another way to switch buffers is to press Ctrl-⟨Tab⟩. This command switches to the buffer you last used. If you press ⟨Tab⟩ again while still holding down Ctrl, you can switch to still older buffers. Hold down Shift as well as Ctrl to move in the reverse order. You can press Ctrl-G to abort and return to the original buffer.

You can also change to another buffer using the next-buffer and previous-buffer commands. They select the next (or previous) buffer and connect it to the current window. You can cycle through all the buffers by repeating these commands. You can type F12 and F11, respectively, to run these commands. If your keyboard doesn't have these keys, you can also type Ctrl-X > and Ctrl-X <.

Summary:	Ctrl-X B	select-buffer
	Ctrl-X Ctrl-B	bufed
	Ctrl-X K	kill-buffer
	Ctrl-X Ctrl-K	kill-current-buffer
	Ctrl-⟨Tab⟩	switch-buffers
	F12, Ctrl-X >	next-buffer
	F11, Ctrl-X <	previous-buffer
		kill-all-buffers
		new-file

4.8.2 Files

Reading Files

The Ctrl-X Ctrl-F key runs the find-file command. It prompts you for a file name. First, it scans the current buffers to see if any of them contain that file. If so, the command connects that buffer to the current window. Otherwise, the command creates a buffer with the same name as the file, possibly modified to make it different from the names of existing non-empty buffers, then reads the file into that buffer. Most people consider find-file the command they typically use to edit a new file, or to return to a file read in previously.

Normally Epsilon examines the file's contents to determine if it's a binary file, or in Unix or Macintosh format. If you prefix a numeric argument to find-file, Epsilon asks you for the correct format, as described on page 100, and for the name of an encoding. (Use the `auto-detect` encoding if you only want to force binary, Unix, Macintosh, or Windows format.)

The find-file command examines a file's name and contents to determine an appropriate language mode for it. For instance, files with a .c extension are put in C mode. You can override this decision with a "file

variable". See page 103. You can use the reset-mode command at any time to make Epsilon repeat that process, setting the buffer to a different mode if appropriate. It can be handy after you've temporarily switched to a different mode for any reason, or after you've started creating a new file with no extension and have now typed the first few lines, enough for Epsilon to auto-detect the proper mode.

If you type ⟨Enter⟩ without typing any file name when find-file asks for a file, it runs dired on the current directory. If you give find-file a file name with wild card characters, or a directory name, it runs the dired command giving it that pattern. See page 110 for a description of the very useful dired command. Also see page 101 for information on related topics like how to type a file name with ⟨Space⟩ characters, customize the way Epsilon prompts for files, and so forth.

By default, at most prompts for file names like find-file's, Epsilon types in for you the directory portion of the current file. For example, suppose the current buffer contains a file named "\src\new\ll.c". If you invoke find-file , Epsilon will type in "\src\new\" for you. This comes in handy when you want to read another file in the same directory as the current file. You can simply begin typing another file name if you want Epsilon to ignore the pre-typed directory name. As soon as Epsilon notices you're typing an absolute file pathname, it will erase the pre-typed directory name. See page 101 for details.

You can change the current directory with the cd command on F7. It prompts for a new current directory, and then displays the full pathname of the selected current directory. You can type the name of a new directory, or just type ⟨Enter⟩ to stay in the current directory. When you supply a file name, Epsilon interprets it with respect to the current directory unless it begins with a slash or backslash. If you specify a drive as part of the directory name, Epsilon will set the current drive to the indicated drive, then switch to the indicated directory. Press Alt-E when prompted for a directory name, and Epsilon will insert the name of the directory containing the current file. (If you start a concurrent process, Epsilon can link its current directory to Epsilon's; see the `use-process-current-directory` variable for details.)

The insert-file command on Ctrl-X I prompts for the name of a file and inserts it before point. It sets the mark before the inserted text, so you can kill it with Ctrl-W. (Also see the `insert-file-remembers-file` variable.)

The find-linked-file command on Ctrl-X Ctrl-L looks for a file name in the current buffer, then finds that file. It works with plain text files, and also understands `#include` in C-like buffers, `<a href=>` in HTML-like buffers, and various other mode-specific conventions. You can highlight a file name first whenever its automatic parsing of file names isn't right. In a process buffer, it looks for error messages, not file names (unless you've first highlighted a file name), and sets the current error message (as used by next-error) to the current line.

Epsilon uses a built-in list of directories to search for `#include` files; you can set the `include-directories` variable to add to that list. For files with a .lst extension, it assumes the current line holds a file name, instead of searching for a pattern that matches a typical file name. This is one way to more easily manage files in a project that are in many different directories.

The key Ctrl-X Ctrl-V runs the visit-file command. It prompts you for a file name. If the file exists, the command reads it into the current buffer, and positions point at the beginning. The command discards the old contents of the buffer, but asks before discarding an unsaved buffer. If no file with the given name exists, the command clears the current buffer. If you prefix this command with a numeric argument, the command discards the old buffer content without warning. So if you want to revert to the copy of the file on disk, disregarding the changes you've made since you last saved the buffer, press Ctrl-U Ctrl-X Ctrl-V, followed by ⟨Enter⟩. Most people use this command only to explicitly manipulate the file associated with a particular buffer. To read in a file, use the find-file command, described above.

The revert-file command rereads the current file from disk. If you've made any unsaved changes, it prompts first.

If a file has an extension `.gz` or `.bz2`, indicating a compressed file, Epsilon automatically decompresses it when you read it. See the `uncompress-files` variable.

Summary: Ctrl-X Ctrl-F   find-file

                           F7   cd

                           Ctrl-X I   insert-file

                           Ctrl-X Ctrl-V   visit-file

                                                    revert-file

Read-Only Files

Whenever you read a read-only file into a buffer using find-file or visit-file, Epsilon makes the buffer read-only, and indicates this by displaying "RO" in the modeline. Epsilon keeps you from modifying a read-only buffer. Attempts to do so result in an error message. In a read-only buffer you can use the ⟨Space⟩ and ⟨Backspace⟩ keys to page forward and back more conveniently; see the `readonly-pages` variable to disable this.

If you want to modify the buffer, you can change its read-only status with the change-read-only command on Ctrl-X Ctrl-Q. With no numeric argument, it toggles the read-only status. With a non-zero numeric argument, it makes the buffer read-only; with a numeric argument of zero, it makes the buffer changeable.

The change-read-only command sets the buffer's status but doesn't change the read-only status of its file. Use the change-file-read-only command to toggle whether or not a file is read-only.

By default, when Epsilon reads a read-only file, it displays a message and makes the buffer read-only. To make Epsilon do something else instead, you can set the `readonly-warning` variable, default 3, according to figure 4.7.

Action	0	1	2	**3**	4	5	6	7
Display a warning message	N	Y	N	**Y**	N	Y	N	Y
Make buffer read-only	N	N	Y	**Y**	N	N	Y	Y
Ring the bell	N	N	N	**N**	Y	Y	Y	Y

Figure 4.7: Values for the readonly-warning variable.

Sometimes you may want to edit a file that is not read-only, but still have Epsilon keep you from making any accidental changes to the file. The find-read-only-file command does this. It prompts for a file name just like find-file and reads it, but marks the buffer read-only so it cannot be modified, and sets it so that if you should ever try to save the file, Epsilon will prompt for a different name.

Summary: Ctrl-X Ctrl-Q   change-read-only

                                                    find-read-only-file

                                                    change-file-read-only

Saving Files

The Ctrl-X Ctrl-S key writes a buffer to the file name associated with the buffer. If the current buffer contains no file, the command asks you for a file name.

To write the buffer to some other file, use the Ctrl-X Ctrl-W key. The command prompts for a file name and writes the buffer to that file. Epsilon then associates that file name with the buffer, so later Ctrl-X Ctrl-S

commands will write to the same file. If the file you specified already exists, Epsilon will ask you to confirm that you wish to overwrite it. To disable this warning, you can set the variable `warn-before-overwrite` to zero. (Setting the variable to zero also prevents several other commands from asking for confirmation before overwriting a file.)

Before Epsilon saves a file, it checks the copy of the file on disk to see if anyone has modified it since you read it into Epsilon. This might happen if another user edited the file (perhaps over a network), or if a program running concurrently with Epsilon modified the file. Epsilon does this by comparing the file's date and time to the date and time Epsilon saved when it read the file in. If they don't match (within a tolerance determined by the `file-date-tolerance` variable), Epsilon displays a warning and asks you what you want to do. You can choose to read the disk version of the file and discard the one already in a buffer, replace the copy on disk with the copy you've edited, or compare the two versions.

Epsilon checks the file date of a file each time you switch to a buffer or window displaying that file, and before you read or write the file. When a file changes on disk and you haven't modified the copy in memory, Epsilon automatically reads the new version. (It doesn't do this automatically if the file on disk is very large, or substantially smaller than the copy in memory.) You can make Epsilon always ask before reading by setting the buffer-specific variable `auto-read-changed-file` to zero.

Set the buffer-specific variable `want-warn` to 0 if you don't want Epsilon to ever check the file date or warn you. Or under Windows, set the `file-date-skip-drives` variable to make Epsilon ignore file dates on specific types of drives, such as network drives or floppy disks.

You can have Epsilon remove any spaces or tabs at the end of each line, before saving a file. See the `c-delete-trailing-spaces` and `default-delete-trailing-spaces` variables.

Epsilon automatically marks a buffer as "modified" when you change it, and shows this with a star '*' at the end of the buffer's mode line. When Epsilon writes a buffer to disk or reads a file into a buffer, it marks the buffer as "unmodified". When you try to exit Epsilon, it will issue a warning if any buffer contains a file with unsaved changes.

You may occasionally want to change a buffer's modified status. You can do this with the change-modified command. Each time you invoke this command, the modified status of the current buffer toggles, unless you invoke it with a numeric argument. A nonzero numeric argument sets the modified status; a numeric argument of zero clears the modified status.

The save-all-buffers command, bound to Ctrl-X S, goes to each buffer with unsaved changes (those marked modified), and if it contains a file, writes the buffer out to that file. See the `save-all-without-asking` variable to alter what Epsilon does when there's an error saving a file.

The write-region command on Ctrl-X W takes the text between point and mark, and writes it to the file whose name you provide.

Summary:	Ctrl-X Ctrl-S	save-file
	Ctrl-X Ctrl-W	write-file
	Alt-˜	change-modified
	Ctrl-X S	save-all-buffers
	Ctrl-X W	write-region

Backup Files

Epsilon doesn't normally keep the previous version of a file around when you save a modified version. If you want backups of saved files, you can set the buffer-specific variable `want-backups` to 1, using the set-variable command described on page 127. If this variable is 1, the first time you save a file in a session,

Epsilon will first preserve the old version by renaming any existing file with that name to a file with the extension ".bak". For instance, saving a new version of the file text.c preserves the old version in text.bak. (If you delete a file's buffer and later read the file again, Epsilon treats this as a new session and makes a new backup copy the next time you save.) If `want-backups` variable is 2, Epsilon will do this each time you save the file, not just the first time.

You can change the name Epsilon uses for a backup file by setting the variable `backup-name`. Epsilon uses this as a *template* for constructing the backup file name. It copies the template, substituting pieces of the original file for codes in the template, according to figure 4.8. The sequence %r substitutes a relative pathname to the original file name, if the file is within the current directory or its subdirectories, or an absolute pathname otherwise.

The sequence %x substitutes the full pathname of the directory containing the Epsilon executable. The sequence %X substitutes the same full pathname, but this time after converting all Windows long file names making up the path to their equivalent short name aliases. For example, if the Epsilon executable was in the directory `c:\Program Files\Eps12\bin\`, %x would use exactly that pathname, while %X might yield `c:\Progra~1\Eps12\bin\`. Under Unix, %X is the same as %x. Either always ends with a path separator character like / or \.

Code	Part	Example 1 c:\dos\read.me	Example 2 /usr/bin
%p	Path	c:\dos\	/usr/
%b	Base	read	bin
%e	Extension	.me	(None)
%f	Full name	c:\dos\read.me	/usr/bin
%r	Relative path	dos\read.me	/usr/bin
	(assuming current dir is	c:\	/usr/mark)
%x	Executable path	c:\Program Files\Eps12\bin\	/usr/local/bin/
%X	Alias to path	c:\Progra~1\Eps12\bin\	/usr/local/bin/

Figure 4.8: File name template characters.

If any other character follows %, Epsilon puts that character into the backup file name. You can use this, for example, to include an actual % character in your backup file name, by putting %% in the template.

Epsilon can automatically save a copy of your file every 500 characters. To make Epsilon autosave, set the variable `want-auto-save` to 1. Epsilon then counts keys as you type them, and every 500 keys, saves each of your modified files to a file with the extension ".asv". Epsilon uses a template (see above) to construct this name as well, stored in the variable `auto-save-name`. You can alter the number of keystrokes between autosaves by setting the variable `auto-save-count`.

Sometimes you may want to explicitly write the buffer out to a file for backup purposes, but may not want to change the name of the file associated with the buffer. For that, use the copy-to-file command on Ctrl-F7. It asks you for the name of a file, and writes the buffer out to that file, but subsequent Ctrl-X Ctrl-S's will save to the original file.

Summary: Ctrl-F7   copy-to-file

Line Translation

Most Windows, DOS and OS/2 programs use files with lines separated by the pair of characters Return, Newline (or Control-M, Control-J). But internally Epsilon separates lines with just the newline character, Ctrl-J. Epsilon normally translates between the two systems automatically when reading or writing text files in this format. When it reads a file, it removes all Ctrl-M characters, and when it writes a file, it adds a Ctrl-M character before each Ctrl-J.

Epsilon will automatically select one of several other translation types when appropriate, based on the contents of the file you edit. It automatically determines whether you're editing a regular file, a binary file, a Unix file, or a Mac file, and uses the proper translation scheme. You can explicitly override this if necessary. Epsilon determines the file type by looking at the first few thousand bytes of the file, and applying heuristics. This is quite reliable in practice. However, Epsilon may occasionally guess incorrectly. You can tell Epsilon exactly which translation scheme to use by providing a numeric argument to a file reading command like find-file, or a file-writing command like save-file or write-file. Epsilon will then prompt for which translation scheme to use.

The set-line-translate command sets this behavior for the current buffer. It prompts for the desired type of translation, and makes future file reads and writes use that translation. Epsilon will display "Binary", "Unix", "DOS", or "Mac" in the mode line to indicate any special translation in effect. (It omits this when the "usual" translation is in effect: Unix files in Epsilon for Unix, DOS files in other versions.)

Set the `default-translation-type` variable if you want to force Epsilon to always use a particular type of translation when reading existing files, rather than examining their contents and choosing a suitable type. Set the `new-buffer-translation-type` variable if you want Epsilon to create new buffers and files with a translation type other than the default. For file names that start with ftp://, the `ftp-ascii-transfers` variable can changes the meaning of some translation types; see its online help.

Epsilon remembers the type of translation you want in each buffer using the buffer-specific variable `translation-type`.

Epsilon applies the following heuristics, in order, to determine a file's type. These may change in future versions.

A file that contains null bytes is considered binary. A file that has no Ctrl-M Ctrl-J pairs is considered a Unix file if it contains Ctrl-J characters, or a Macintosh file if it contains Ctrl-M. A file containing a Ctrl-M character not followed by either Ctrl-M or Ctrl-J is considered binary. Any other files, or files of less than five characters, are considered to be in standard DOS format (in Epsilon for Unix, Unix format).

Bear in mind that Epsilon makes all these decisions after examining only the first few thousand bytes of a file, and phrases like "contains null bytes" really mean "contains null bytes in its first few thousand characters."

Summary: set-line-translate

DOS/OEM Character Set Support

Windows programs typically use a different character set than do DOS programs, or programs that run in a Win32 console environment. The DOS character set is known as the DOS/OEM character set, and includes various line drawing characters and miscellaneous characters not in the Windows/ANSI set. The Windows/ANSI character set includes many accented characters not in the DOS/OEM character set. Epsilon for Windows uses the Windows/ANSI character set (with most fonts). Epsilon for Win32 Console uses a DOS/OEM character set by default, but see the `console-ansi-font` variable.

The oem-to-ansi command converts the current buffer from the DOS/OEM character set to the Windows/ANSI character set. The ansi-to-oem command does the reverse. If any character in the buffer doesn't have a unique translation, these commands warn before translating, and move to the first character without a unique translation.

The find-oem-file command reads a file using the DOS/OEM character set, translating it into the Windows/ANSI character set, and arranges things so when you save the file, the reverse translation automatically occurs.

The commands in this section provide a subset of the functionality available with the Unicode-based commands described on page 135. The oem-to-ansi command is similar to the unicode-convert-from-encoding command. Specify an encoding such as "cp850" or "cp437", using the code page number shown by the "chcp" command at a Windows command prompt. Similarly, the ansi-to-oem command is like the unicode-convert-to-encoding command. The find-oem-file command is like invoking find-file with a numeric prefix argument, so it asks for line translation and encoding options, and specifying the DOS/OEM encoding as above. See page 135 for details on setting a default code page and similar options.

Summary: oem-to-ansi
                                                     ansi-to-oem
                                                     find-oem-file

File Name Prompts

You can customize many aspects of Epsilon's behavior when prompting for file names.

By default, many commands in the Windows version of Epsilon use the standard Windows common file dialog, but only when you invoke them from a menu or the tool bar. When you invoke these commands using their keyboard bindings, they use the same kind of dialog as other Epsilon prompts.

Set `want-common-file-dialog` to 2 if you want Epsilon to use the common file dialog whenever it can. Set `want-common-file-dialog` to 0 to prevent Epsilon from ever using this dialog. The default value of 1 produces the behavior described above.

The Windows common file dialog includes a list of common file extensions. You can customize this list by editing the file filter.txt, putting your own version in your customization directory (see page 11). See the comments in that file for more information. You can also customize which directory this dialog uses, and how Epsilon remembers that choice; see the `common-open-use-directory` variable.

All the remaining variables described in this section have no effect when Epsilon uses the standard Windows dialog; they only modify Epsilon's own file dialogs.

The `prompt-with-buffer-directory` variable controls how Epsilon uses the current directory at file prompts. When this variable is 2, the default, Epsilon inserts the current buffer's directory at many file prompts. This makes it easy to select another file in the same directory. You can edit the directory name, or you can begin typing a new absolute pathname right after the inserted pathname. Epsilon will delete the inserted pathname when it notices your absolute pathname. This behavior is similar to Gnu Emacs's.

When `prompt-with-buffer-directory` is 1, Epsilon temporarily changes to the current buffer's directory while prompting for a file name, and interprets file names relative to the current directory. This behavior is similar to the "pathname.e" extension available for previous versions of Epsilon.

When `prompt-with-buffer-directory` is 0, Epsilon doesn't do anything special at file prompts. This was Epsilon's default behavior in previous versions.

The grep and file-query-replace commands use a separate variable
grep-prompt-with-buffer-directory for their file patterns, with the same meaning as above. By
default it's 1.

During file name completion, Epsilon can ignore files with certain extensions. The
ignore-file-extensions variable contains a list of extensions to ignore. By default, this variable has the
value '|.obj|.exe|.b|.b2|.bu|', which makes file completion ignore files that end with .obj, .exe, .b,
.b2, and .bu. Each extension must appear between '|' characters. You can augment this list using the
set-variable command, described on page 127.

Similarly, the only-file-extensions variable makes completion look only for files with certain
extensions. It uses the same format as ignore-file-extensions, a list of extensions surrounded by |
characters. If the variable holds a null pointer, Epsilon uses ignore-file-extensions as above.

When Epsilon prompts for a file name, the ⟨Space⟩ key performs file name completion on what you've
typed. To create a new file with spaces in its name, you must quote the space characters by typing Ctrl-Q
before each one, while entering the name, or type " characters around the file name (or any part containing
spaces).

At any Epsilon prompt (not just file prompts), you can type Alt-E to retrieve your previous response to
that prompt. Alt-⟨Up⟩ or Ctrl-Alt-P show a list of previous responses. See page 26 for complete details.
Alt-⟨Down⟩ or Ctrl-Alt-N let you easily copy text from the buffer into the prompt (useful when the buffer
contains a file name or URL). See page 24 for more information. At most file name prompts, Alt-G will
retrieve the name of the current buffer's file.

When Epsilon shows a dialog containing a list of previous responses, or files matching a pattern, the list
may be too wide for the dialog. You can generally resize the dialog by simply dragging its border. This
works for most Epsilon dialogs. Epsilon will automatically remember the size of each dialog from session to
session.

File Name Case

When retrieving file names from some file systems, Epsilon automatically translates the file names to lower
case. Epsilon uses various different rules for determining when to convert retrieved file names to lower case,
and when two file names that differ only by case refer to the same file.

Epsilon distinguishes between three types of file systems:

On a case-sensitive file system, MyFile, MYFILE, and myfile refer to three different files. Unix file
systems are case-sensitive.

On a case-preserving (but not case-sensitive) file system, MyFile, MYFILE, and myfile all refer to the
same file. But if you create a file as MyFile, the file system will display that file as MyFile without altering
its case. VFAT, NTFS, and HPFS file systems used in Windows and OS/2 are case-preserving.

On a non-case-preserving file system, MyFile, MYFILE, and myfile all refer to the same file. Moreover,
the operating system converts all file names to upper case. So no matter how you create the file, the
operating system always shows it as MYFILE. DOS's FAT file system is non-case-preserving. When
Epsilon displays a file name from such a file system, it changes the file name to all lower case.

Epsilon for Windows asks the operating system for information on each drive, the first time the drive is
accessed. Epsilon for Unix assumes all file systems are case-sensitive, and the rest of this section does not
apply.

You can tell Epsilon to use particular rules for each drive on your system by defining an environment
variable. The MIXEDCASEDRIVES environment variable should contain a list of drive letters or ranges. If
the variable exists and a lower case letter like k appears in it, Epsilon assumes drive K: has a Unix-style
case-sensitive file system. If the variable exists and an upper case letter like J appears in it, Epsilon assumes

drive J: is not case-preserving or case-sensitive, like traditional FAT drives. If the variable exists but a drive letter does not appear in it, Epsilon assumes the drive has a case-preserving but not case-sensitive file system like NTFS, HPFS, or VFAT drives.

If, for example, drives h:, i:, j:, and p: access Unix filesystems over a network, drive q: accesses a server that uses a FAT filesystem, and other drives use a VFAT filesystem (local drives under Windows, for example), you could set MIXEDCASEDRIVES to `h-jpQ`. When Epsilon finds a MIXEDCASEDRIVES variable, it assumes the variable contains a complete list of such drives, and doesn't examine filesystems as described. If an EPSMIXEDCASEDRIVES configuration variable exists, that overrides any MIXEDCASEDRIVES environment variable that may be found. (Note that MIXEDCASEDRIVES appears in the environment under all operating systems, while EPSMIXEDCASEDRIVES is a configuration variable must be put in the registry under Windows. See page 8 for details.)

You can set the variable `preserve-filename-case` nonzero to tell Epsilon to use the case of filenames exactly as retrieved from the operating system. By default, Epsilon changes all-uppercase file names to lower case, except on case-sensitive file systems. The variable also controls case conversion rules when Epsilon picks a buffer name for a file, and related settings.

File Variables

The find-file command examines a file's name and contents to determine an appropriate language mode for it. For instance, files with a .c extension are put in C mode. You can override this decision with a "file variable".

These are specially-formatted lines at the top or bottom of a file that indicate the file should use a particular language mode or tab size. For example, you can put `-*- mode:  VBasic -*-` anywhere on the first line of a file to force Epsilon to Visual Basic mode, or write `-*- tab-size:  3 -*-` to make Epsilon use that tab size setting.

Epsilon recognizes a syntax for file variables that's designed to be generally compatible with Emacs. The recognized formats are as follows. First, the first line of the file (or the second, if the first starts with `#!`, to accommodate the Unix "shebang" line) may contain text in one of these formats:

`-*- mode:` *modename* `-*-`

`-*-` *modename* `-*-`

`-*- tab-width:` *number* `-*-`

`-*- mode:` *modename*`; tab-width:` *number* `-*-`

Other characters may appear before or after each possibility above; typically there would be commenting characters, so a full line might read `/* -*- mode:  shell -*- */`. The first two examples set that buffer to the specified mode name, such as Perl or VBasic or C, by running a command named *modename*-mode if one exists. (A mode name of "C++" makes Epsilon uses the C++ submode of C mode.) The third example sets the width of a tab character for that buffer.

In more detail, between the `-*-` sequences may be one or more definitions, separated by `;` characters. Spacing and capitalization are ignored throughout. Each definition may either be a mode name alone, or a setting name followed by a colon `:` and a value.

The setting names recognized are "mode", as another way to specify the mode; "tab-size" or "tab-width" to set the buffer's tab size, or "margin-right" or "fill-column" to set the buffer's right margin. (The names `tab-size` and `margin-right` reflect the names of the Epsilon variables they set; the names "tab-width" and "fill-column" are more compatible with other programs, and recommended if non-Epsilon users may edit the files.) Epsilon also recognizes "compile-command" for use with the compile-buffer

command; see page 123 for details. And it recognizes "coding" to indicate the file's Unicode encoding, if the `detect-encodings` variable permits this.

The other syntax for file variables must appear at the end of the file, starting within the last 3000 characters. It looks like this:

```
Local Variables:
mode: modename
tab-size: number
End:
```

The first and last lines are required; inside are the settings, one per line. Each line may have additional text at the start and end of each line (so it will look like a comment in the file's programming language). The "coding" file variable doesn't use this alternative syntax; any specified encoding must be on the first line only.

4.8.3   Internet Support

Epsilon for Windows or Unix has several commands and facilities that make it easy for you to edit files on other computers using the Internet.

The find-file and dired commands, as well as a few others, understand Internet URL's. If you provide the URL ftp://user@example.com/myfile.c to a file-reading command like find-file, Epsilon will engage in an FTP interaction to download the file and display it in a buffer. All of the Internet activity happens in the background, so you don't have to wait for the file to download before continuing with your work. In fact, the file appears in the buffer as it downloads (syntax highlighted if appropriate), so you can be editing the beginning of a large file while the rest of it downloads.

Saving a file in such a buffer, or writing a buffer to a file name that starts with ftp://, will cause Epsilon to send the file to the remote computer. Upload and download status is indicated in the mode line, and there's also a show-connections command (on Ctrl-Alt-C) that shows the status of all Internet activities and buffers. As in bufed, you can select a buffer and press ⟨Enter⟩ to switch to it, or press ⟨Escape⟩ to remain in the current buffer. Use the kill-process command to cancel an FTP transfer or Telnet session (see below) in progress in the current buffer.

FTP and SCP URL's (the latter described in the next section) work with dired also, so if you do a dired (or a find-file) on ftp://user@example.com, you'll get a directory listing of the files on the remote machine `example.com`, in a familiar dired context. Dired knows how to delete and rename remote files, and sort by size, date, file name or extension. To make Epsilon work with certain host computers (systems running VMS, for example), you may need to set the variables `ftp-ascii-transfers` or `ftp-compatible-dirs` nonzero; see the descriptions of those variables in the online help. Other systems may require you to set the variable `ftp-passive-transfers`.

The telnet command lets you connect to a command shell on a remote computer. It puts you in a buffer that works much like the Epsilon process buffer, except the commands you type are executed on the remote machine. Provide a numeric prefix argument and telnet will connect on the specified port instead of the default port. Or use the syntax `hostname:port` for the host name to specify a different port. You can either use the telnet command directly, or specify a telnet: URL to find-file. (Epsilon ignores any username or password included in the URL.) Also see the ssh command described in the next section.

In a telnet or ssh buffer, the `telnet-interpret-output` variable controls whether Epsilon interprets certain ANSI color-setting escape sequences and similar things. Epsilon also looks for password requests from the remote system, using the `recognize-password-pattern` variable, so it can hide the password as you type it.

As in a concurrent process buffer, you can press Alt-P or Alt-N to access a telnet buffer's command history. With a numeric prefix argument, these keys show a menu of all previous commands. You can select one to repeat.

If you specify an http: URL to find-file (for example, http://www.lugaru.com), Epsilon will use the HTTP protocol to retrieve the HTML code from the given location. The HTML code will appear in an appropriately named buffer, syntax highlighted. Header information for the URL will be appended to a buffer named "HTTP Headers".

You can tell Epsilon to send its requests by way of a proxy by setting the variables `http-proxy-server`, `http-proxy-port`, and `http-proxy-exceptions`. You can tell Epsilon to identify itself to the server as a different program by setting `http-user-agent`, or set `http-force-headers` to entirely replace Epsilon's HTTP request with another. The `http-log-request` variable makes Epsilon copy the entire request it sends to the HTTP Headers buffer.

The Alt-E and Alt-⟨Down⟩ keys in find-file come in handy when you want to follow links in an HTML buffer; see page 26 for information on Alt-E and page 24 for information on Alt-⟨Down⟩. Also see the find-linked-file command on Ctrl-X Ctrl-L.

The command view-web-site on Shift-F8 searches for the next URL in the buffer. It prompts with that URL, and after you modify it if necessary, it then launches an external browser on the URL. The view-lugaru-web-site command launches a browser and points it to Lugaru's web site. Epsilon for Unix uses a shell script named `goto_url` to run a browser. See page 35. Epsilon for Windows uses the system's default browser.

The finger command prompts for a string like "user@example.com", then uses the finger protocol to query the given machine for information about the given user. The output appears in an appropriately named buffer.

If you run a compiler via telnet or a similar process in an Epsilon buffer, you can set up the next-error command on Ctrl-X Ctrl-N so that when it parses a file name in an error message, it translates it into a URL-style file name that Epsilon can use to access the file. To do this, you'll need to write your own `telnet_error_converter()` subroutine in EEL. See the sample one in the Epsilon source file epsnet.e for details.

Summary: Ctrl-Alt-C   show-connections
              Telnet mode only: Alt-n   process-next-cmd
              Telnet mode only: Alt-p   process-previous-cmd
                                                      telnet
                                                      telnet-mode
                                                      finger
                                                      view-web-site
                                                      view-lugaru-web-site

Secure Shell and SCP Support

Besides recognizing ftp:// URL's, Epsilon also recognizes scp:// URL's, which may be used for secure file transfers. With scp support, you can read or write files using an scp:// URL, navigate the remote system's directory tree using dired, mark files for copying between the local and remote systems, and use file name completion.

Epsilon recognizes ssh:// URL's to connect securely to a command shell on a remote computer, providing a secure alternative to the telnet command. Epsilon's ssh command works similarly to the ssh:// URL. Use the syntax `username@hostname` to connect as a user other than the default one.

The scp and ssh features work by running certain external programs. Epsilon's `ssh` command depends on an external ssh program. With default settings, scp features use an sftp program, which is part of the ssh package. For Linux or FreeBSD, install the appropriate ssh package for your distribution.

For Windows, the Cygwin system contains appropriate clients, or with some configuration you can use alternatives like PuTTY. (See below for more on PuTTY.) PuTTY works better if you need to type passwords or passphrases and don't use an ssh-agent, though certain features like file name completion won't be available. Existing Cygwin users may prefer its ssh package instead; its main drawback at this writing is that password and passphrase prompts may not appear. Scp and ssh support works on Unix and under Windows NT/2000/XP, but not under Windows 95/98/ME.

With scp/ssh support, Epsilon doesn't prompt for or pass along any passwords or passphrases. You should use an ssh agent to manage your keys and passphrases (see the manual page for ssh-agent in your ssh distribution), or use an authentication method that doesn't require you to type a password (in environments where that's appropriate).

If you don't do either of these things, you may have to type your password or passphrase multiple times, especially if you begin a second scp operation before the first has completed, but in other circumstances too. If the scp/ssh client prompts for a password or passphrase in a buffer, Epsilon will display a message like "Password requested in buffer scp-helper-sample–1". You can type the password by switching to that buffer and using the Alt-x `send-invisible` command, but this method isn't very convenient, and won't work with some Windows ssh clients. Using an ssh-agent as above is much better.

The variable `ssh-template` tells Epsilon how to build a command line for invoking the external ssh program when a specific user name appears before the host name. If no user name was specified, it uses `ssh-no-user-template`. See the descriptions of these variables for their format.

Epsilon runs an sftp program to copy files, obtain file listings for dired, and do many miscellaneous tasks like create directories or perform completion. Some old sftp programs use a different command syntax for listing files; if you have trouble, try setting the `scp-client-style` variable to 2 to make Epsilon use old-style sftp commands.

If you must use a very old version of ssh that lacks an sftp program, or connect to a system that doesn't support sftp, or you want to use an ssh replacement that lacks sftp, you can set up Epsilon to run its own helper program on the remote system.

To do this, copy the C language source code file `epsilon-xfer-helper.c` included in Epsilon's source directory to the remote system, compile it with "make epsilon-xfer-helper" or similar, and install in an accessible location. It may be compiled on most Unix systems, or, for Windows, using the Cygwin environment. Next, check that you can run the helper program remotely, with a command line like

```
ssh -l username hostname epsilon-xfer-helper
```

It should print a greeting line and await a command. Type ˆC or press ⟨Enter⟩ to make it exit. You may need to edit the Epsilon variable `scp-run-helper-template` to include the path to the helper program, or if you use a different ssh program. For instance, if you use an ssh client "oldssh" that lacks an sftp program, set it to "oldssh %u@%h /path/to/epsilon-xfer-helper" or similar. (Epsilon uses the above variable when the scp:// url includes a user name, and the `scp-run-helper-no-user-template` variable when it does not.)

To tell Epsilon to use epsilon-xfer-helper commands, not sftp commands, set the `scp-client-style` variable to 1. Using the helper program enables a few minor features that the sftp program doesn't currently support, like using ˜ to indicate home directories, or copying a remote file to a different location on the remote system (sftp can rename remote files but not copy them).

When you don't use sftp, Epsilon must run a separate program for each file transfer. By default it uses the scp program. The variable `scp-read-file-template` tells Epsilon how to transfer a file from the remote system to a local file, and `scp-write-file-template` does the opposite. There are separate

versions of these variables for when no user name is included, named `scp-read-file-no-user-template` and `scp-write-file-no-user-template`. Change these variables to use a different program for copying files when you don't use sftp.

To make Epsilon work with the Windows ssh client PuTTY, use these settings:

scp-run-helper-no-user-template	`psftp %h`
scp-run-helper-template	`psftp %u@%h`
ssh-template	`plink -l %u %h`
ssh-no-user-template	`plink %h`
scp-client-style	`2`

Because PuTTY's psftp client currently doesn't support wildcards in file listings, some functions like completion won't work.

It's possible to set up Epsilon to use one set of variables for one remote system and a different one for others. To enable this, before checking for a variable such as `scp-run-helper-template`, Epsilon constructs a new variable name by adding the host name of the remote system to its end. For instance, if you try to access www.example.com, Epsilon first looks for a variable named `scp-run-helper-template-www-example-com`; if there's a variable by that name, Epsilon uses it instead of the usual one. (Epsilon constructs the variable name from a host name by replacing each non-alphanumeric character with a -.) It does this for each of the scp and ssh variables mentioned above.

Summary: ssh
                                                 ssh-mode

URL Syntax

In Epsilon, URL's must start with ftp://, http://, scp:// or telnet://. (If you omit the service name, the ftp: part, Epsilon for Windows will pass the file name to Windows as a UNC-style network file name.)

For some services, you can specify a user name, password, or port number using the URL syntax of service://username:password@hostname:portnumber/filepath. (Ftp and http recognize all three, telnet recognizes only a port number, and scp recognizes only a user name.)

If you include a user name in an ftp or http URL but omit the :password part, Epsilon will prompt for one (and will make sure the password does not appear in your state file, session file, or similar places). But if you include a password in your URL, note that it may be saved in Epsilon's session file or similar places.

If you omit the username:password@ or username@ part entirely in an ftp URL, Epsilon uses the user name "anonymous" and the password specified by the `anon-ftp-password` variable (default: EpsilonUser@unknown.host). You can set this to your email address if you prefer.

You can also use Emacs-style syntax for specifying remote file names: /username@hostname:filepath. Epsilon will behave as if you had typed the corresponding URL.

In ftp:// URL's, Epsilon treats a file name following the / as a relative pathname. That is, ftp://user@example.com/myfile refers to a file named myfile in the user's home directory. Put two slashes, as in ftp://user@example.com//myfile, to refer to /myfile in the root directory. You can type \ instead of / in any URL and Epsilon will substitute /.

If you type the name of a local directory to the find-file command, find-file will run the dired command on it. With ftp:// URL's, find-file won't always know that what you typed is a remote directory name (as opposed to a file name) and might try to retrieve the URL as a file, leading to an error message like "`Not a plain file`". End your URL with a / to indicate a directory name.

4.8.4 Printing

The print-buffer command on Alt-F9 prints the current buffer. If a region is highlighted on the screen, the command prints just that region. The print-region command on Shift-F9 always prints just the current region, whether or not it's highlighted.

Under Windows, the printing commands display the familiar Windows print dialog. From this dialog, you can select a different printer, select particular pages to print, and so forth. The print-setup command lets you select a different printer without printing anything, or set the margins. Invoke the printing commands with a numeric prefix argument to skip the print dialog and just print with default settings. The print-buffer-no-prompt command also skips the print dialog and uses default settings.

You can change the font Epsilon for Windows uses for printing with the set-printer-font command. See page 90 for more information.

By default, Epsilon for Windows will print in color on color printers, and in black & white on non-color printers. You can set the `print-in-color` variable to 0, if you don't want Epsilon to ever print in color, or to 2 if you want Epsilon to attempt to use colors even if the printer doesn't appear to be a color printer. (Some printers will substitute shades of grey.) The default value, 1, produces color printing only on color printers.

If you have a color printer, and want to use a different color scheme when printing than you do for screen display, set the variable `print-color-scheme` to the name of the color scheme Epsilon should use for printing.

Epsilon for Windows prints a heading at the top of each page. You can set the `print-heading` variable to control what it includes. The value 1 makes Epsilon include the file name, 2 makes Epsilon include a page number, and 4 makes Epsilon include the current date. You can add these values together; the default value of 7 includes all the above items.

You can set the variable `print-line-numbers` nonzero if you want Epsilon to include line numbers, or set `print-doublespaced` if you want Epsilon for Windows to skip alternate lines.

In non-Windows environments, the printing commands prompt for the device name of a printer. They then write the text to that device name. If you want Epsilon to run a program that will print the file, you can do that too. See the description of the `print-destination` variable in the online help. (For Unix, see `print-destination-unix`, which by default runs the `lpr` program to print a file.) If you want Epsilon for Windows to run a program in order to print a file, bypassing the Windows print dialog, you can set `want-gui-printing` to zero.

By default, Epsilon converts tabs to spaces in a copy of the buffer before printing it. Set the variable `print-tabs` to one if you want Epsilon to print the file just as it is, including the tab characters.

Summary:	Alt-F9	print-buffer
	Shift-F9	print-region
		print-setup

4.8.5 Extended file patterns

This section describes Epsilon's extensions to the rules for wildcard characters in file names. You can specify more complicated file name patterns in Epsilon than Windows or Unix normally allow, using the wildcard characters of square brackets [], commas, semicolons, and curly braces {}. Epsilon also lets you use the * and ? characters in more places. These patterns work in the grep command, the dired command, and in all other places where file name wildcards make sense. (They don't work with Internet URL's, though.)

First, you can put text after the standard wildcard character * and Epsilon will match it. In standard DOS-style patterns, the system ignores any text in a pattern between a * and the end of the pattern (or the dot before an extension). But in Epsilon, `ab*ut` matches all files that start with `ab` and end with `ut`. The * matches the dot character in file names, so the above pattern matches file names like `about` as well as `absolute.out`. (Use `ab*ut.` to match only files like the former, or `ab*.*ut` to match ones like the latter.)

Instead of `?` to match any single character (except dot, slash, or backslash), you can provide a list of characters in square brackets (similar to the regular expression patterns of searching). For example, `file[0123456789stuvw]` matches `file4`, `file7`, and `files`, but not `filer`. Inside the square brackets, two characters separated by a dash represent a range, so you could write the above pattern as `file[0-9s-w]`. A caret character `^` just after the `[` permits any character but the listed ones, so `fil[^tm]er` matches all the files that `fil?er` matches, except `filter` and `filmer`. (To include a dash or `]` in the pattern, put it right after the `[` or `^`. The pattern `[^-]]` matches all characters but `-` and `]`.)

You can use `?` and `*` (and the new square bracket syntax) in directory names. For example, `\v*\*.bat` might match all `.bat` files in `\virtmem` and in `\vision`. Because a star character never matches backslash characters, it would not match `\vision\subdir\test.bat`.

The special directory name `**` matches any number of directory names. You can use it to search entire directory trees. For example, `\**\*.txt` matches all `.txt` files on the current drive. The pattern `**\include\*.h` matches all `.h` files inside an `include` directory, looking in the current directory, its subdirectories, and all directories within those. A pattern ending in `**` matches all files in that hierarchy.

The simplest new file pattern character is the comma. You can run grep on the file pattern `foo,bar,baz` and Epsilon will search in each of the three files. You can use a semicolon in place of a comma, if you want.

A segment of a file pattern enclosed in curly braces may contain a sequence of comma-separated parts. Epsilon will substitute each of the parts for the whole curly-brace sequence. For example, `\cc\include\c*t.{bat,txt}` matches the same files as `\cc\include\c*t.bat,\cc\include\c*t.txt`. A curly-brace sequence may not contain another curly-brace sequence, but may contain other wildcard characters. For example, the pattern `{,c*\}*.{txt,bat}` matches `.txt` and `.bat` files in the current directory, or in any subdirectory starting with "c". The brace syntax is simply a shorthand for the comma-separated list described above, so that an equivalent way to write the previous example is `*.txt,c*\*.txt,*.bat,c*\*.bat`. Epsilon breaks a complete pattern into comma-separated sections, then replaces each section containing curly braces with all the possible patterns constructed from it. You can use semicolons between the parts in braces instead of commas if you prefer.

To match file names containing one of the new wildcard characters, enclose the character in square brackets. For example, the pattern `abc[}]` matches the file name `abc}`. (Note that legal DOS file names may not contain any of the characters `[],;`, but they may contain curly braces `{}`. Other file systems, including Windows VFAT, Windows NT's NTFS, most Unix file systems, and OS/2's HPFS, allow file names that contain any of these characters.)

Use curly braces to search on multiple drives. `{c,d,e}:\**\*.txt` matches all `.txt` files on drives C:, D:, or E:. Epsilon does not recognize the `*`, `?`, or `[]` characters in the drive name.

When a file name contains a literal brace character, a comma, or one of the other characters used for extended wildcard patterns, you can surround it in quotes (`"`) to tell Epsilon to treat it literally, not as a wildcard pattern. Or you can set the `file-pattern-wildcards` variable to disable the wildcarding function of specific characters. If your file names often contain commas, for instance, you may want to disable comma's wildcard function.

It's possible to make Epsilon ignore certain types of symbolic links when interpreting file patterns under Unix. For instance, you can keep a `**` pattern from matching symbolic links to directories, only matching actual directories. See the `file-pattern-rules` variable.

4.8.6   Directory Editing

Epsilon has a special mode used for examining and changing the contents of a directory conveniently. The dired command, bound to Ctrl-X D, asks for the name of a directory and puts a listing of the directory, similar to what the DOS "dir" command produces (or, for Unix, "ls -lF"), in a special dired buffer. By default, dired uses the current directory. You can supply a file pattern, such as "*.c", and only matching files will appear. The dired command puts the information in a buffer whose name matches the directory and file pattern, then displays the buffer in the current window. You can have multiple dired buffers, each displaying the result of a different file pattern.

You can also invoke dired from the find-file command. If you press ⟨Enter⟩ without typing any file name when find-file asks for a file, it does a dired on the current directory. If you give find-file a file name with wild card characters, it runs the dired command giving it that pattern. If you give find-file a directory name, it does a dired of that directory. (When using ftp:// URL's that refer to a directory, end them with /. See page 107 for details.)

You can use extended file patterns to list files from multiple directories. (See page 108.) If you use a file pattern that matches files in more than one directory, Epsilon will divide the resulting dired buffer into sections. Each section will list the files from a single directory. Epsilon sorts each section separately.

While in a dired buffer, alphabetic keys run special dired commands. See the next section on page 110 for a complete list.

The quick-dired-command command on Alt-o is like running a dired on the current file, then executing a single dired command and discarding the dired buffer. It provides a convenient way of performing various simple file operations without running dired. It prompts for another key, one of C, D, M, G, !, T, or V. Then it (respectively) copies, deletes, or renames the current file, changes Epsilon's current directory to the one containing that file, runs a command on the file, shows the file's properties, or views it using associations. Alt-o + creates a new directory, prompting for its name. Alt-o . displays a dired of the current file. Alt-o A lets you set the file's attributes or permission bits. Alt-o F views its folder in MS-Windows Explorer. The other keys are similar to their corresponding dired subcommands; see the next section for more details. (The T and F options are only available in Epsilon for Windows.)

By default, Epsilon records dired buffers in its session file and recreates them the next time you start Epsilon, except for remote direds that use a URL. See the variables `session-restore-directory-buffers` and `session-restore-max-directories`.

The locate-file command prompts for a file name and then searches for that file, using dired to display the matches. In Windows, it searches for the file on all local hard drives, skipping over removable drives, CD-ROM drives, and network drives. On Unix, it searches through particular parts of the directory hierarchy specified by the `locate-path-unix` variable.

The list-files command also takes a file pattern and displays a list of files. Unlike dired, its file list uses absolute pathnames, and it omits the file's size, date, and other information. It provides just the file names, one to a line. The command also doesn't list directory names, as dired does. The command is often useful when preparing response files for other programs.

Summary:	Ctrl-X D	dired
	Alt-o	quick-dired-command
		list-files

Dired Subcommands

This section lists the subcommands you can use when editing a dired buffer (see page 110). You run most dired commands by pressing plain letters. All other keys still invoke the usual Epsilon commands.

The `N` and `P` commands go to the next and previous files, respectively.

The `E`, ⟨Space⟩, and ⟨Enter⟩ keys let you examine the contents of a file. They invoke the find-file command on the file, making the current window display this file instead of the dired buffer. To conveniently return to the dired buffer, use the select-buffer command (Ctrl-X B). Press ⟨Enter⟩ when prompted for the buffer name and the previous buffer shown in the current window (in this case, the dired buffer) will reappear.

When applied to a subdirectory, these keys invoke another dired on that directory, using the name of the directory for that dired buffer. If you have marked files for deletion or copying, and you run a dired on the same directory, the markings go away.

The '.' or "^" keys invoke a dired on the parent directory of the directory associated with the current dired buffer.

To set Epsilon's current directory to the directory being displayed, press `G` (for Go). If the current line names a directory, Epsilon will make that be the current directory. If the current line names a file, Epsilon will set the current directory to the one containing that file.

Press `D` to flag a file that you wish to delete. Epsilon will mark the file for deletion by placing a 'D' before its name. (You may delete empty directories in the same way.) Press `C` or `M` to select files for copying or moving (renaming), respectively. Epsilon will mark the files by placing `C` or `M` before their names. The `U` command unmarks the file on the current line, removing any marks before its name.

The `X` command actually deletes, copies, or moves the marked files. Epsilon will list all the files marked for deletion and ask you to confirm that you want them deleted. If any files are marked for copying or moving, Epsilon will ask for the destination directory into which the files are to be copied or moved. If there is only one file to copy or move, you can also specify a file name destination, so you can use the command for renaming files. (In this case, Alt-g will copy the original file name so you can edit it.) Epsilon prompts for a single destination for all files to be copied, and another for all files to be moved.

If you try to delete a read-only file, Epsilon will prompt first; see the `dired-confirmation` variable to change this. If you try to delete a non-empty directory, Epsilon will similarly ask for confirmation before deleting the entire directory hierarchy.

There are a few specialized commands for renaming files. Press Shift-L to mark a file for lowercasing its name, or Shift-U for uppercasing. When you execute with X, each marked file will be renamed by changing each uppercase character in its name to lowercase (or vice versa). (Note that Epsilon for Windows displays all-uppercase file names in lowercase by default, so Shift-U's effect may not be visible within Epsilon. See `preserve-filename-case`.)

Shift-R marks a file for a regular-expression replacement on its name. When you press X to execute operations on marked files, Epsilon will ask for a pattern and replacement text. Then, for each file marked with Shift-R, Epsilon will take the file name and perform the indicated regular expression replacement on it, generating a new name. Then Epsilon will rename the file to the new name. For instance, to rename a group of files like dir\file1.cxx, dir\file2.cxx, etc. to dir2\file1.cpp, dir2\file2.cpp, use Shift-R and specify `dir\(.*).cxx` as the search text and `dir2\#1.cpp` as the replacement text. To rename some .htm files to .html, specify `.*` as the search text and `#0l` as the replacement text.

By default, most files or directories that start with a period character `.` will be hidden. Pressing – toggles whether such files are hidden. The `dired-show-dotfiles` variable sets which files or directories are always shown regardless of this toggle. By default, dired entries for the current directory (`.`) and its parent (`..`) are always shown.

The `!` dired subcommand prompts for a command line, then runs the specified program, adding the name of the current line's file after it. If the command line you type contains an `*`, Epsilon substitutes the current file name at that position instead of at the end. If the command line ends in a `&` character, Epsilon runs the program asynchronously; otherwise it waits for the program to finish.

The + command creates a new subdirectory. It asks for the name of the subdirectory to create.

The R command refreshes the current listing. Epsilon will use the original file pattern to rebuild the file listing. If you've marked files for copying, moving, or deleting, the markings will be discarded if you refresh, so Epsilon will prompt first to confirm that you want to do this.

The S key controls sorting. It prompts you to enter another letter to change the sorting method. Press N, E, S, or D to select sorting by file name, file extension, size, or time and date of modification, respectively. Press U to turn off sorting the next time Epsilon makes a dired listing, and display the file names in the same order they come from the operating system. (You can have Epsilon rebuild the current listing using the R subcommand.)

Press + or – at the sorting prompt to sort in ascending or descending order, respectively, or R to reverse the current sorting order.

Press G at the sorting prompt to toggle directory grouping. With directory grouping, Epsilon puts all subdirectories first in the list, then all files, and sorts each part individually. Without directory grouping, it mixes the two together (although it still puts . and .. first).

Under Windows, press A to display the file's current attributes (Hidden, System, Read-only and Archive) and specify a new attribute list. You can set the dired-layout variable under Windows to include these attributes in the dired listing itself. Under Unix, A runs the chmod command, passing it the mode specification you type, such as g+w to let group members write to the file.

Press V to run the "viewer" for that file; the program assigned to it according to Windows file associations. For Windows executable files, this will run the program. For document files, it typically runs the Windows program assigned to that file extension. See page 115 for information on associating Epsilon with particular file extensions.

Under Unix, V uses Gnome file associations to run the viewer for the file. Run the gnomecc program to select a different viewer.

Press Shift-P to print the current file. Under Windows, press T to display the properties of a file or directory. (This is a convenient way to see the total size of all files in a directory.)

Several keys provide shortcuts for common operations. The 1 key examines the selected file in a window that occupies the whole screen (like typing Ctrl-X 1 E). The 2 key splits the current window horizontally and examines the selected file in the second window, leaving the dired buffer in the first (like typing Ctrl-X 2 E). The 5 key functions like the 2 key, but splits the window vertically (like typing Ctrl-X 5 E). The 0 key examines the selected file in the next window on the screen, without splitting windows any further. The Z key zooms the window to full-screen, then examines the selected file (like typing Ctrl-X Z E).

Press lowercase L to create a live link. First Epsilon creates a second window, if there's only one window to start with. (Provide a numeric argument to get vertical, not horizontal, window splitting.) Then Epsilon displays the file named on the current dired line in that window, in a special live link buffer. As you move around in the dired buffer, the live link buffer will automatically update to display the current file. Files over dired-live-link-limit bytes in size won't be shown, to avoid delays. Delete the live link buffer or window, or show a different buffer there, to stop the live linking.

Finally, typing H or ? while in dired displays help on these dired subcommands.

4.8.7   Buffer List Editing

The bufed command on Ctrl-X Ctrl-B functions like dired, but it works with buffers instead of files. It creates a list of buffer names. Each buffer name appears on a line along with the size of the buffer, the associated file name (if any) and a star if the buffer contains unsaved changes, and/or an R if the buffer is currently marked read-only. The bufed command pops up the list, and highlights the line describing the current buffer.

In this buffer, alphabetic keys run special bufed commands. Alphabetic keys not mentioned do nothing, and non-alphabetic keys run the usual commands. The N and P keys go to the next and previous buffers in the list, respectively, by going down or up one line. The D command deletes the buffer on the current line, but warns you if the buffer contains unsaved changes. The S key saves the buffer on the current line, and Shift-P prints the buffer like the print-buffer command. The E or ⟨Space⟩ command selects the buffer on the current line and displays it in the current window, removing the bufed listing.

As in dired, several keys provide shortcuts for common operations. The 1 key expands the current window to take up the whole screen, then selects the highlighted buffer. The 2 key splits the current window horizontally and selects the highlighted buffer in the second window. The 5 key works like the 2 key, except it splits the window vertically. The Z key zooms the current window to full-screen, then selects the highlighted buffer.

By default, the most recently accessed buffers appear at the top of the list, and those you haven't used recently appear at the end. The current buffer always appears at the top of the list. You can press 'b', 'f', or 'i' to make Epsilon sort the list by buffer name, file name, or size, respectively. Pressing 'a' makes Epsilon sort by access time again. Pressing the upper case letters 'B', 'F', 'I', or 'A' reverses the sense of the sort. Pressing 'u' produces a buffer list ordered by time of creation, with the oldest buffers at the bottom.

The bufed command does not normally list special buffers such as the kill buffers, whose names start with a dash character ("–"). To include even these buffers, give the bufed command a numeric argument. By default, bufed pops up a 50-column window in the non-Windows versions. You can change this width by setting the `bufed-width` variable. (In Epsilon for Windows, change the dialog's width by dragging its border, as usual.) See the `bufed-show-absolute-path` variable to change the form of each file name that bufed displays.

Summary: Ctrl-X Ctrl-B   bufed

4.9   Starting and Stopping Epsilon

You generally exit the editor with Ctrl-X Ctrl-Z, which runs the command exit-level. If in a recursive editing level, exit-level will not exit, but bring you back to the level that invoked the recursive edit. If you haven't saved all your files, Epsilon will display a list using bufed and ask if you really want to exit.

You may also use exit, Ctrl-X Ctrl-C, to exit the editor. It ignores any recursive editing levels. When given a numeric argument, Epsilon won't warn you about unsaved files, write a session file (see the next section), record your current font settings, or similar things. It will simply exit immediately.

You can customize Epsilon's actions at startup by defining a hook function using EEL. See page 299.

In Epsilon for Unix, an alternative to exiting Epsilon is to suspend it using the Alt-x suspend-epsilon command. This returns control to the shell that launched Epsilon. Use the shell's fg command to resume Epsilon. When Epsilon runs as an X program, this command instead minimizes Epsilon's window.

Summary: Ctrl-X Ctrl-Z   exit-level
                 Ctrl-X Ctrl-C   exit
                                                  suspend-epsilon

4.9.1   Session Files

When you start up Epsilon, it will try to restore the window and buffer configuration you had the last time you ran Epsilon. It will also restore items such as previous search strings, your positions within buffers, and

the window configuration. The `-p` flag described on page 13 or the `preserve-session` variable may be used to disable restoring sessions.

You can set the `session-restore-max-files` variable to limit the number of files Epsilon will reread, which is by default 15. The files are prioritized based on the time of their last viewing in Epsilon, so by default Epsilon restores the 15 files you've most recently edited. Also, Epsilon won't automatically restore any files bigger than the size in bytes specified by the `session-restore-biggest-file` variable. For files accessed via URL's, Epsilon uses the variable `session-restore-biggest-remote-file` instead.

By default, Epsilon records dired buffers (see page 110) in its session file and recreates them the next time you start Epsilon, except for remote direds that use a URL. Set the variables `session-restore-directory-buffers` or `session-restore-max-directories` to customize this.

You can set the `session-restore-directory` variable to control whether Epsilon restores any current directory setting in the session file. Set it to 0 and Epsilon will never do this. Set it to 1 and Epsilon will always restore the current directory when it reads a session file. The default value 2 makes Epsilon restore the current directory setting only when the `-w1` flag has been specified. (Under Windows, Epsilon's installer includes this flag when it makes Start Menu shortcuts.)

If you set the variable `session-always-restore` to zero, Epsilon will only try to restore your previous session if you invoke it without giving a file name on the command line. If you provide an explicit file to edit on the command line, Epsilon will read just that file in, and will refrain from restoring the previous session. (Also see page 7.)

You can set the `session-restore-files` variable to control whether Epsilon restores files named in a session file, or just search strings, command history, and similar settings. If `session-restore-files` is 0, when Epsilon restores a session, it won't load any files named in the session, only things like previous search strings. If 1, the default, Epsilon will restore previous files as well as other settings. If 2, Epsilon will restore previous files only if there were no files specified on Epsilon's command line.

The write-session command writes a session file, detailing the files you're currently editing, the window configuration, default search strings, and so forth. By default, Epsilon writes a session file automatically whenever you exit, but you can use this command if you prefer to save and restore sessions manually.

The read-session command loads a session file, first asking if you want to save any unsaved files. Reading in a session file rereads any files mentioned in the session file, as well as replacing search strings, all bookmarks, and the window configuration. However, any files not mentioned in the session file will remain, as will keyboard macros, key bindings, and most variable settings. If you use either command and specify a different session file than the default, Epsilon will use the file name you provided when it automatically writes a session file as you exit.

Locating The Session File

By default, Epsilon restores your previous session by consulting a single session file named epsilon.ses, which is normally stored in your customization directory. (See page 11.) Epsilon will write such a file when you exit.

You can set Epsilon to use multiple sesssion files by having it search for an existing session file, starting from the current directory. If a session file doesn't exist in the current directory, then Epsilon looks in its parent directory, then in that directory's parent, and so forth, until it reaches the root directory or finds a session file. Or you can have it always read and create its session file in the current directory.

To make Epsilon look for its session file only in the current directory, and create a new session file there on exiting, set the `session-default-directory` variable to ".".

To make Epsilon search through a directory hierarchy for an existing session file, set the `session-tree-root` variable to empty. If this variable is set to a directory name in absolute form, Epsilon will only search for an existing session file in the named directory or one of its children. For example, if `session-tree-root` holds c:\joe\proj, and the current directory is c:\joe\proj\src, Epsilon will search in c:\joe\proj\src, then c:\joe\proj, for a session file. If the current directory is c:\joe\misc, on the other hand, Epsilon won't search at all (since \joe\misc isn't a child of \joe\proj), but will use the rules below. By default this variable is set to the word `NONE`, an impossible absolute directory name, so searching is disabled.

If Epsilon finds no such file by searching as described above (or if such searching is disabled, as it usually is), then Epsilon looks for a session file in each of these places, in this order:

- If the `session-default-directory` variable is non-empty, in the directory it names. (This variable is empty by default.)

- If the configuration variable EPSPATH can be found, in the first directory it names. (See page 8 for more on configuration variables.)

- In your customization directory.

There are three ways to tell Epsilon to search for a file with a different name, instead of the default of epsilon.ses. With any of these methods, specifying an absolute path keeps Epsilon from searching and forces it to use a particular file. Epsilon checks for alternate names in this order:

- The -p flag can specify a different session file name.

- An ESESSION configuration variable can specify a different session file name.

- The `session-file-name` variable can specify a name.

Summary: read-session
                                                            write-session

4.9.2   File Associations and DDE

You can set up file associations in Epsilon for Windows using the create-file-associations command. It lets you modify a list of common extensions, then sets up Windows to invoke Epsilon to edit files with those extensions. The files will be sent to an existing copy of Epsilon, if one is running, via a Windows DDE execute message.

Dynamic Data Exchange, or DDE, is one mechanism in Windows for programs to talk to each other. A DDE server is a program that knows how to "listen" for messages from other programs. A DDE client is a program that knows how to send a message using DDE.

When you double-click on a shell icon, and you want a program to start editing a file, Windows arranges for this in one of two ways. The simple way: Windows just starts a new copy of the program and tells it to edit that file. The disadvantage is that you get multiple copies of the program running, if you click on multiple files.

The better way uses DDE. The create-file-associations command sets things up so that Windows will know how to use DDE to talk to a copy of Epsilon. So now when you double-click on a file registered to Epsilon, Windows will first try to send a message to Epsilon saying "please edit this file". If there's no running copy of Epsilon, Windows will notice that no program accepted the message, and it will know it needs to run the program itself.

Summary: create-file-associations

4.9.3   Sending Files to a Prior Session

Epsilon's command line flag −add tells Epsilon to locate an existing instance of itself (a "server"), send it a message containing the rest of the command line, and immediately exit. (Epsilon ignores the flag if there's no prior instance.) This feature works in Epsilon for Windows and Epsilon for Unix.

The command line flag −noserver tells Epsilon that it should not respond to such messages from future instances.

The command line flag −server may be used to alter the server name for an instance of Epsilon, which is "Epsilon" by default. An instance of Epsilon started with −server:*somename* −add will only pass its command line to a previous instance started with the same −server:*somename* flag.

An −add message to Epsilon uses a subset of the syntax of Epsilon's command line. It can contain file names to edit, the +*linenum* flag, the flag −d*varname*=*value* to set an Epsilon variable, −l*filename* to load an EEL bytecode file, or −r*funcname* to run an EEL function, command, or macro. Use −r*funcname*=*arg* to run an EEL function and pass it a single string parameter *arg*. Epsilon unminimizes and tries to move to the foreground whenever it gets a message, unless the message uses the −r flag and doesn't include a file name.

Spaces separate file names and flags in the message; surround a file name or flag with " characters if it contains spaces. In EEL, such messages arrive via a special kind of `WIN_DRAG_DROP` event.

In Epsilon for Unix you can use the −wait flag instead of −add. This causes the client Epsilon to send the following command line to an existing instance and then wait for a response from the server, indicating the user has finished editing the specified file. Use the resume-client command on Ctrl-C # to indicate this.

Epsilon for Windows normally acts as a server for its own internal-format messages, as described above, and also acts as a DDE server for messages from Windows Explorer. The −noserver flag described above also disables DDE, and the −server flag also sets the DDE server name. The DDE server in Epsilon uses a topic name of "Open" and a server name determined as described above (normally "Epsilon").

Summary: Ctrl-C #   resume-client

4.9.4   MS-Windows Integration Features

Epsilon can integrate with Microsoft's Visual Studio (Developer Studio) in several ways. The on-demand style of integration lets you press a key (or click a button) while editing a file in Visual Studio, and start Epsilon on the same file. The other automates this process, so any attempt to open a source file in Visual Studio is routed to Epsilon.

For on-demand integration, you can add Epsilon to the Tools menu in Microsoft Visual Studio. You'll then be able to select Epsilon from the menu and have it begin editing the same file you're viewing in Visual Studio, at the same line.

To do this in Visual Studio 5.0 or 6.0, use the Tools/Customize menu command in Visual Studio. Select the Tools tab in the Customize dialog that appears. Create a new entry for the Tools menu, and set the Command field to the name of Epsilon's executable, epsilon.exe. Include its full path, typically c:\Program Files\Eps12\Bin\epsilon.exe. Set the Arguments field to `-add +$(CurLine):$(CurCol) $(FilePath)`. Set the Initial Directory field to `$(FileDir)`. After creating this new command, you can then use the Tools/Customize/Keyboard command to set up a shortcut key for it. You may also want to configure Visual

Studio to detect when a file is changed outside its environment, and automatically load it. See Tools/Options/Editor for this setting.

If you use Visual Studio .NET, the steps are slightly different. Use Tools/External Tools/Add to create a new entry in the Tools menu. Set the Command field to the full path to Epsilon's executable, epsilon.exe, as above. Set the Arguments field to `-add +$(CurLine):$(CurCol) $(ItemPath)`. Optionally, set the Initial Directory field to `$(ItemDir)`. You can use Tools/Options/Environment/Keyboard to set up a shortcut key for the appropriate Tools.ExternalCommand entry. To set Visual Studio .NET to autoload modified files, use Tools/Options/Environment/Documents.

You can also set up Visual Studio 5.0 or 6.0 to do the above automatically, so that every time Visual Studio tries to open a source file, Epsilon appears and opens the file instead. To set up Visual Studio so its attempts to open a source file are passed to Epsilon, use the Customize command on the Tools menu and select the Add-ins and Macro Files page in the dialog. Click Browse, select Add-ins (.dll) as the File Type, and navigate to the VISEPSIL.DLL file located in the directory containing Epsilon's executable (typically `c:\Program Files\Eps12\bin`). Select that file.

Close the Customize dialog and a window containing an Epsilon icon (a blue letter E) should appear. You can move the icon to any toolbar by dragging it. Click the icon and a dialog will appear with two options. Unchecking the first will disable this add-in entirely. If you uncheck the second, then any time you try to open a text file in Dev Studio it will open in both Epsilon and Dev Studio. When checked, it will only open in Epsilon.

Running Epsilon via a Shortcut

Epsilon comes with a program, sendeps.exe, that's installed in the directory containing Epsilon's main executable. It provides some flexibility when you create a desktop icon for Epsilon, or use the Send To feature (both of which involve creating a Windows shortcut).

If you create a desktop shortcut for Epsilon, or use the Send To feature in Windows, have it refer to this sendeps.exe program instead of Epsilon's main executable. Sendeps will start Epsilon if necessary, or locate an existing copy of Epsilon, and load the files named on its command line.

This is useful because Windows ignores a shortcut's flags (command line settings) when you drop a document on a shortcut, or when you use the Send To feature. (If it used the flags, you could simply create a shortcut to Epsilon's main executable and pass its -add flag. Since it doesn't, sending a file requires a separate program.) Also, Windows sends long file names without quoting them in these cases, which would cause problems if sent directly to Epsilon.

Sendeps may be configured through entries in a lugeps.ini file located in your Windows directory. The section name it uses is the same as the base name of its executable (so making copies of the executable under different names lets you have multiple Send To entries that behave differently, for instance).

These are its default settings:

```
[SendEps]
server=Epsilon
topic=Open
ddeflags=
executable=epsilon.exe
runflags=-add -w1
nofilestartnew=1
nofileflags=-w1
usedde=0
```

Here's how Sendeps uses the above settings. It first looks for an Epsilon server named `server` using Epsilon's `-add` protocol. If found, it sends the server a command line consisting of the `ddeflags` setting, followed by the file name passed on its command line (inside double quotes). If there's no such server running, Sendeps executes a command line built by concatenating the `executable` name, the `runflags`, and the quoted file name.

You can tell Sendeps to use DDE instead of its usual `-add` protocol by setting usedde to 1. In that case it will use the specified `topic` name.

When you invoke Sendeps without specifying a file name on its command line, its behavior is controlled by the `nofilestartnew` setting. If nonzero it starts a new instance of Epsilon. If zero, it brings an existing instance to the top, if there is one, and starts a new instance otherwise. In either case, if it needs to start a new instance it uses `nofileflags` on the command line.

The Open With Epsilon Shell Extension

If you tell Epsilon's installer to add an entry for Epsilon to every file's context menu in Explorer, Epsilon installs a shell extension DLL. You can configure it by creating entries in the lugeps.ini file located in your Windows directory.

These are its default settings, which you can copy to lugeps.ini as a basis for your changes:

```
[OpenWith]
server=Epsilon
serverflags=
executable=epsilon.exe
runflags=-add -w1
menutext=Open With Epsilon
```

When you select Open With Epsilon from the menu in Explorer, the shell extension first looks for an Epsilon server named `server` using Epsilon's `-add` protocol. If found, it sends the server a command line consisting of the `serverflags` setting, followed by the file name you selected (inside double quotes).

If there's no such server running, the DLL executes a command line built by concatenating the `executable` name, the `runflags`, and the quoted file name. If the `executable` name is a relative pathname, it first tries to run any executable by that name located in the DLL's current directory. If that fails, it uses the executable name as-is, and lets Windows search for it along the PATH.

If you've selected multiple files, it repeats the above process for each file.

You can alter the menu text Explorer displays by setting the `menutext` item. This setting doesn't take effect until you restart Explorer, or unload and reload the `owitheps.dll` file that provides this menu by running `regsvr32 /u owitheps.dll`, then `regsvr32 owitheps.dll`. Other changes to the DLL's settings take effect immediately.

4.10   Running Other Programs

Epsilon provides several methods for running other programs from within Epsilon. The `push` command on Ctrl-X Ctrl-E starts a command processor (shell) running. You can then issue shell commands. When you type the "exit" command, you will return to Epsilon and can resume your work right where you left off.

With a numeric argument, the command asks for a command line to pass to the shell, runs this command, then returns.

While Epsilon runs a command processor or other program with the push command, it looks like you ran the program from outside of Epsilon. But Epsilon can make a copy of the input and output that occurs during the program's execution, and show it to you when the program returns to Epsilon. If you set the variable capture-output to a nonzero value (normally it has the value zero), Epsilon will make such a transcript. When you return to Epsilon, this transcript will appear in a buffer named "process".

You can use the filter-region command on Alt-| to process the current region through an external command. Epsilon will run the command, sending a copy of the region to it as its standard input. By default, the external command's output goes to a new buffer. Run filter-region with a numeric argument if you want the output to replace the current region.

The command shell-command is similar, but it doesn't send the current region as the command's input. It prompts for the name of an external program to run and displays the result in a buffer; with a numeric argument it inserts the command's output into the current buffer.

Configuration variables (see page 8) let you customize what command Epsilon runs when it wants to start a process. Epsilon runs the command file named by the EPSCOMSPEC configuration variable. If no such variable exists, Epsilon uses the standard COMSPEC environment variable instead. Epsilon reports an error if neither exists.

If a configuration variable named INTERSHELLFLAGS has been defined, Epsilon passes the contents of this variable to the program as its command line. When Epsilon needs to pass a command line to the program, it doesn't use INTERSHELLFLAGS. Instead, it inserts the contents of the CMDSHELLFLAGS variable before the command line you type. (The sequence %% in CMDSHELLFLAGS makes Epsilon interpolate the command line at that point, instead of adding it after the flags.)

If Epsilon can't find a definition for INTERSHELLFLAGS or CMDSHELLFLAGS, it substitutes flags appropriate for the operating system.

Summary: Ctrl-X Ctrl-E   push
                                                            filter-region
                                                            shell-command

4.10.1   The Concurrent Process

Epsilon can also run a program in a special way that allows you to interact with the program in a buffer and continue editing while the program runs. It can help in preparing command lines, by letting you edit things you previously typed, and it automatically saves what each program types, so you can examine it later. If a program takes a long time to produce a result, you can continue to edit files while it works. We call a program run in this way a *concurrent process*.

The start-process command, bound to Ctrl-X Ctrl-M, begins a concurrent process. Without a numeric argument, it starts a shell command processor which will run until you exit it (by going to the end of the buffer and typing "exit"). With a numeric argument, it creates an additional process buffer, in Epsilon environments that support more than one.

Epsilon maintains a command history for the concurrent process buffer. You can use Alt-P and Alt-N to retrieve the text of previous commands. With a numeric prefix argument, these keys show a menu of all previous commands. You can select one to repeat.

In a concurrent process buffer, you can use the ⟨Tab⟩ key to perform completion on file names and command names as you're typing them. If no more completion is possible, it displays all the matches in the echo area, if they fit. If not, press ⟨Tab⟩ again to see them listed in the buffer. See the process-complete command for more details, or the process-completion-style variable to customize how process buffer completion works.

Under Windows NT/2000/XP, certain process buffer functions like completion, or interpreting compiler error messages, require Epsilon to determine the current directory of the command processor running there. Epsilon does this by examining each prompt from cmd.exe, such as `C:\WINNT>`. If you've set a different format for the prompt, you may have to set the `process-prompt-pattern` variable to tell Epsilon how to retrieve the directory name from it. Also see the `use-process-current-directory` variable to change how Epsilon's current directory and the process's are linked together.

As described in the previous section, you can change the name of the shell command processor Epsilon calls, and specify what command line switches Epsilon should pass to it, by setting configuration variables. Some different configuration variable names override those variables, but only when Epsilon starts a subprocess concurrently. For example, you might run a command processor that you have to start with a special flag when Epsilon runs it concurrently. The INTERCONCURSHELLFLAGS and CMDCONCURSHELLFLAGS variables override INTERSHELLFLAGS and CMDSHELLFLAGS, respectively. The EPSCONCURCOMSPEC variable overrides EPSCOMSPEC.

For example, one version of the Bash shell for Windows systems requires these settings:

```
EPSCOMSPEC=c:\cygwin\bin\bash.exe
INTERSHELLFLAGS=--login --noediting -i
CMDSHELLFLAGS=--login -c "%%"
```

These are configuration variables, so they would go in the environment for Epsilon for Unix, or in the system registry for Windows versions. See page 8.

When a concurrent process starts, Epsilon creates a buffer named "process". In this buffer, you can see what the process types and respond to the process's requests for input. If a buffer named "process" already exists, perhaps from running a process previously, Epsilon goes to its end. Provide a numeric argument to the start-process command and it will create an additional process buffer (in those environments where Epsilon supports multiple process buffers).

If you set the variable `clear-process-buffer` to 1, the commands start-process, push, and make (described below) will each begin by emptying the process buffer. The variable normally has a value of 0. (See that variable for more options.) Set the variable `start-process-in-buffer-directory` to control which directory the new process starts in.

A program running concurrently behaves as it does when run directly from outside Epsilon except when it prints things on the screen or reads characters from the keyboard. When the program prints characters, Epsilon inserts these in the process buffer. When the program waits for a line of input, Epsilon will suspend the process until it can read a line of input from the process buffer, at which time Epsilon will restart the process and give it the line of input. You can type lines of input before the program requests them, and Epsilon will feed the input to the process as it requests each line. In some environments, Epsilon will also satisfy requests from the concurrent process for single-character input.

In detail, Epsilon remembers a particular spot in the process buffer where all input and output takes place. This spot, called the *type point*, determines what characters from the buffer a program will read when it does input, and where the characters a program types will appear. Epsilon inserts in the buffer, just before the type point, each character a program types. When a process requests a line of input, Epsilon waits until a newline appears in the buffer after the type point, then gives the line to the program, then moves the type point past these characters. (In environments where Epsilon can distinguishes a request by a program to read a single character, it will pause the concurrent process until you have inserted a character after the type point, give that character to the concurrent process, then advance the type point past that character.)

You may insert characters into the process buffer in any way you please, typing them directly or using the yank command to retrieve program input from somewhere else. You can move about in the process buffer, edit other files, or do anything else at any time, regardless of whether the program has asked the system for keyboard input.

To generate an end-of-file condition for DOS or Windows programs reading from the standard input, insert a ^Z character by typing Ctrl-Q Ctrl-Z on a line by itself, at the end of the buffer. For a Unix program, type Ctrl-Q Ctrl-D ⟨Enter⟩.

Some programs will not work when running concurrently. Programs that do cursor positioning or graphics will not work well, since such things do not correspond to a stream of characters coming from the program to insert into a buffer. They may even interfere with what Epsilon displays. We provide the concurrent process facility primarily to let you run programs like compilers, linkers, assemblers, filters, etc.

There are some limitations on the types of programs you can run under Epsilon for Windows 95/98/ME. Specifically, 32-bit Win32 console mode programs running concurrently under Epsilon for Windows 95/98/ME cannot receive console input. These restrictions don't apply under Windows NT/2000/XP.

If you run Epsilon under Windows 95/98/ME, you may find it necessary to increase the environment space available to a subprocess. To do this, locate the file conagent.pif in the directory containing Epsilon's executable (typically `c:\Program Files\Eps12\bin`). (Explorer may be set to hide the file's .pif extension.) Display its properties, and on the Memory tab enter a value in bytes for the Initial Environment setting.

Under Windows 95/98/ME, Epsilon will let you run only one other program at a time. Under Unix, or other versions of Windows, you may rename the buffer named "process" using the rename-buffer command, and start a different, independent concurrent process in the buffer "process". Or run the start-process command with a numeric argument to have Epsilon pick a unique buffer name for the new process buffer if "process" already has an active process. If you exit Epsilon while running a concurrent process, Epsilon kills that process.

The exit-process command types "exit" to a running concurrent process. If the concurrent process is running a standard command processor, it should then exit. Also see the `process-warn-on-exit` and `process-warn-on-killing` variables.

The kill-process command disconnects Epsilon from a concurrent process, and forces it to exit. It operates on the current buffer's process, if any, or on the buffer named "process" if the current buffer has no process. If the current buffer isn't a process buffer but has a running Internet job (such as an ftp:// buffer), this command tries to cancel it.

The stop-process command, normally on Ctrl-C Ctrl-C, makes a program running concurrently believe you typed Control-Break (or, for Unix, sends an interrupt signal). It operates on the current buffer's process, if any, or on the buffer named "process" if the current buffer has no process.

Summary:	Ctrl-X Ctrl-M	start-process
	Ctrl-C Ctrl-C	stop-process
	Process mode only: Alt-⟨Backspace⟩	process-backward-kill-word
	Process mode only: ⟨Tab⟩	process-complete
	Process mode only: C-Y	process-yank
	Process mode only: Alt-n	process-next-cmd
	Process mode only: Alt-p	process-previous-cmd
		kill-process
		exit-process

4.10.2   Compiling From Epsilon

Many compilers produce error messages in a format that Epsilon can interpret with its next-error command on Ctrl-X Ctrl-N. The command searches in the process buffer (beginning at the place it reached last time,

or at the beginning of the last command) for a line that contains a file name, a line number, and an error message. If it finds one, it uses the find-file command to retrieve the file (if not already in a window), then goes to the appropriate line in the file. With a numeric argument, it finds the *n*th next error message, or the *n*th previous one if negative. In particular, a numeric argument of 0 repeats the last message. The previous-error command on Ctrl-X Ctrl-P works similarly, except that it searches backward instead of forward.

The Ctrl-X Ctrl-N and Ctrl-X Ctrl-P keys move back and forth over the list of errors. If you move point around in a process buffer, it doesn't change the current error message. You can use the find-linked-file command on Ctrl-X Ctrl-L to reset the current error message to the one shown on the current line. (The command also goes to the indicated source file and line, like Ctrl-X Ctrl-N would.)

Actually, Ctrl-X Ctrl-N runs the next-position command, not next-error. The next-position command usually calls next-error. After you use the grep command (see page 41), however, next-position calls next-match instead, to move to the next match of the pattern you searched for. If you use any command that runs a process, or run next-error explicitly, then next-position will again call next-error to move to the next error message.

Similarly, Ctrl-X Ctrl-P actually runs previous-position, which decides whether to call previous-error or previous-match based on whether you last ran a compiler or searched across files.

To locate error messages, the next-error command performs a regular-expression search using a pattern that matches most compiler error messages. See page 55 for an explanation of regular expressions. The command uses the ERROR_PATTERN macro and others, defined in the file proc.e. You can change these patterns if they don't match your compiler's error message format. The next-error command also uses another regular-expression pattern to filter out any error messages Epsilon should skip over, even if they match ERROR_PATTERN. The variable ignore-error stores this regular expression. For example, if ignore-error contains the pattern ".*warning", Epsilon will skip over any error messages that contain the word "warning".

The next-error command knows how to use the Java CLASSPATH to locate Java source files, and has special logic for running Cygwin-based programs under Windows. (See the cygwin-filenames variable for more information on the latter.) If you run a compiler via telnet or a similar process in an Epsilon buffer, you can set up next-error to translate file names in the telnet buffer into URL-style file names that Epsilon can use to access the file. See page 105.

The command view-process on Shift-F3 can be convenient when there are many long error messages in a compilation. It pops up a window showing the process buffer and its error messages, and lets you move to a particular line with an error message and press ⟨Enter⟩. It then goes to the source file and line in error. You can also use it to see the complete error message from the compiler, when next-error's one-line display is inadequate.

The make command on Ctrl-X M runs a program and scans its output for error messages using next-error. In some environments, it runs the program in the background, displaying a "Compiling" message while it runs. If you don't start other editing before it finishes, it automatically goes to the first error message. If you do, it skips this step; you can press Ctrl-X Ctrl-N as usual to go to the first error. In other environments, Epsilon may run the program in a concurrent process buffer, or non-concurrently. See the variables compile-in-separate-buffer and concurrent-make for more details.

By default, it runs a program called "make", but with a numeric argument it will prompt for the command line to execute. It will use that command line from then on, if you invoke make without a numeric argument. See the variable start-make-in-buffer-directory to control which directory the new process starts in.

Epsilon uses a template for the command line (stored in the push-cmd variable), so you can define a command line that depends on the current file name. See page 99 for information on templates. For example, cl %f runs the cl command, passing it the current file name.

If a concurrent process already exists, Epsilon will attempt to run the program concurrently by typing its name at the end of the process buffer (in those environments where Epsilon isn't capable of creating more than one process buffer). When Epsilon uses an existing process buffer in this way, it will run next-error only if you've typed no keys during the execution of the concurrent program. You can set the variable `concurrent-make` to 0 to force Epsilon to exit any concurrent process, before running the "make" command. Set it to 2 to force Epsilon to run the command concurrently, starting a new concurrent process if it needs to. When the variable is 1 (the default), the make command runs the compiler concurrently if a concurrent process is already running, non-concurrently otherwise.

Whenever push or make exit from a concurrent process to run a command non-concurrently, they will restart the concurrent process once the command finishes. Set the `restart-concurrent` variable to zero if you don't want Epsilon to restart the concurrent process in this case.

Before make runs the program, it checks to see if you have any unsaved buffers. If you do, it asks if it should save them first, displaying the buffers using the bufed command. If you say yes, then the make command saves all of your unsaved buffers using the save-all-buffers command (which you can also invoke yourself with Ctrl-X S). You can modify the `save-when-making` variable to change this behavior. If it has a value of 0, Epsilon won't warn you that you have unsaved buffers. If it has a value of 1, Epsilon will automatically save all the buffers without asking. If it has a value of 2 (as it has normally), Epsilon asks.

The compile-buffer command on Alt-F3 is somewhat similar to make, but tries to compile only the current file, based on its extension. There are several variables like `compile-cpp-cmd` you can set to tell Epsilon the appropriate compilation command for each extension. If Epsilon doesn't know how to compile a certain type of file, it will prompt for a command line. While Epsilon's make command is good for compiling entire projects, compile-buffer is handy for compiling simple, one-file programs.

The command is especially convenient for EEL programmers because compile-buffer automatically loads the EEL program into Epsilon after compiling it. The EEL compiler is integrated into Epsilon, so Epsilon doesn't need to run another program to compile. When Epsilon compiles EEL code using its internal EEL compiler, it looks in the `compile-eel-dll-flags` variable for EEL command line flags.

The buffer-specific `concurrent-compile` variable tells compile-buffer whether to run the compiler concurrently. The value 2 means always run the compiler concurrently, 0 means never run concurrently, and 1 means run concurrently if and only if a concurrent process is already running. The value 3 (the default) means use the value of the variable `concurrent-make` instead. (The `concurrent-make` variable tells the make command whether to run its program concurrently, and takes on values of 0, 1, or 2 with the same meaning as for `concurrent-compile`.)

A file can use a file variable named "compile-command" (see page 103) to tell compile-buffer to use a specific command to compile that file, not the usual one implied by its file name extension. The command line must be surrounded with " characters if it contains a semicolon. For instance,

```
-*- compile-command: "gcc -I/usr/local/include %r" -*-
```

on the first line of a file tells Epsilon to use that command to compile that file. Unlike other file variables, Epsilon doesn't scan for a compile command when you first load the file; it does this each time you use the compile-buffer command. Also see the variable `use-compile-command-file-variable`.

Summary:		
	Ctrl-X Ctrl-N	next-position
	Ctrl-X Ctrl-P	previous-position
		next-error
		previous-error
	Shift-F3	view-process
	Ctrl-X M	make
	Alt-F3	compile-buffer

4.11 Repeating Commands

4.11.1 Repeating a Single Command

You may give any Epsilon command a numeric prefix argument. Numeric arguments can go up to several hundred million, and can have either a positive or negative sign. Epsilon commands, unless stated otherwise in their description, use a numeric argument as a repetition count if this makes sense. For instance, forward-word goes forward 10 words if given a numeric argument of 10, or goes backward 3 words if given a numeric argument of -3.

The argument command, normally bound to Ctrl-U, specifies a numeric argument. After typing Ctrl-U, type a sequence of digits and then the command to which to apply the numeric argument. Typing a minus sign changes the sign of the numeric argument.

You may also use the Alt versions of the digit keys (Alt-1, etc.) with this command. (Note that by default the numeric keypad keys plus Alt do not give Alt digits. They produce keys like Alt-⟨PgUp⟩ or let you enter special characters by their numeric code. See the `alt-numpad-keys` variable.) You can enter a numeric argument by holding down the Alt key and typing the number on the main keyboard. Alt-⟨Minus⟩ will change the sign of a numeric argument, or start one at -4.

If you omit the digits, and just say Ctrl-U Ctrl-F, for instance, Epsilon will provide a default numeric argument of 4 and move forward four characters. Typing another Ctrl-U after invoking argument multiplies the current numeric argument by four, so typing Ctrl-U Ctrl-U Ctrl-N will move down sixteen lines. In general typing a sequence of n Ctrl-U's will produce a numeric argument of 4^n.

The run-with-argument command provides an alternative way to run a command with a numeric argument. It prompts for the argument with a normal Epsilon numeric prompt, so that you can yank the number to use from the clipboard, or specify it with a different base like 0x1000 (for hexadecimal), or specify the number as a character code like `'q'` or `'\n'` or `<Yen Sign>`.

Summary: Ctrl-U   argument
                                                            run-with-argument

4.11.2 Keyboard Macros

Epsilon can remember a set of keystrokes, and store them away in a *keyboard macro*. Executing a keyboard macro has the same effect as typing the characters themselves. Use keyboard macros to make repetitive changes to a buffer that involve the same keystrokes. You can even write new commands with keyboard macros.

To define a keyboard macro, use the Ctrl-X (command. The echo area will display the message "Remembering", and the word "Def" will appear in the mode line. Whatever you type at the keyboard gets executed as it does normally, but Epsilon also stores the keystrokes away in the definition of the keyboard macro.

When you have finished defining the keyboard macro, press the Ctrl-X) key. The echo area will display the message "Keyboard macro defined", and a keyboard macro named last-kbd-macro will then exist with the keys you typed since you issued the Ctrl-X (command. To execute the macro, use the Ctrl-F4 command (or use Ctrl-X E if you prefer). This executes the last macro defined from the keyboard. If you want to repeatedly execute the macro, give the Ctrl-F4 command a numeric argument telling how many times you want to execute the macro.

You can bind this macro to a different key, naming it as well, using the bind-last-macro function on Ctrl-X Alt-N. Once the macro has its own name, defining a new macro won't overwrite it. This command prompts for a new name, then asks for a key binding for the macro. (You can press Ctrl-G at that point if you want to give the macro a name but not its own key binding.) The name-kbd-macro command prompts for a name but doesn't offer to bind the macro to a key.

You can make a keyboard macro that suspends itself while running to wait for some user input, then continues. Press Shift-F4 while writing the macro and Epsilon will stop recording. Press Shift-F4 again to continue recording. When you play back the macro, Epsilon will stop at the same point in the macro to let you type in a file name, do some editing, or whatever's appropriate. Press Shift-F4 to continue running the macro. When a macro has been suspended, "Susp" appears in the mode line.

Keyboard macros do not record most types of mouse operations. Commands in a keyboard macro must be keyboard keys. However, you can invoke commands on a menu or tool bar while defining a keyboard macro, and they will be recorded correctly. While running a macro, Epsilon's commands for killing and yanking text don't use the clipboard; see page 51.

Instead of interactive definition with Ctrl-X (, you can also define keyboard macros in a command file. The details appear in the section on command files, which starts on page 131. Command files also provide a way to edit an existing macro, by inserting it into a scratch buffer in an editable format with the insert-macro command, modifying the macro text, then using the load-buffer command to load the modified macro.

Epsilon doesn't execute a keyboard macro as it reads the definition from a command file, like it does when you define a macro from the keyboard. This causes a rather subtle difference between the two methods of definition. Keyboard macros may contain other keyboard macros, simply by invoking a second macro inside a macro definition. When you create a macro from the keyboard, the keys you used to invoke the second macro do not appear in the macro. Instead, the text of the second macro appears. This allows you to define a temporary macro, accessible with Ctrl-F4, and then define another macro using the old macro.

With macros defined from files, this substitution does not take place. Epsilon makes such a macro contain exactly the keys you specified in the file. When you execute this macro, the inner macro will execute at the right time, then the outer macro will continue, just as you would expect.

The difference between these two ways of defining macros that contain other macros shows up when you consider what happens if you redefine the inner macro. An outer macro defined from the keyboard remains the same, since it doesn't contain any reference to the inner macro, just the text of the inner macro at the time you defined the outer one. However, an outer macro defined from a file contains a reference to the inner macro, by name or by a key bound to that macro. For this reason the altered version of the inner macro will execute in the course of executing the outer macro.

Normally Epsilon refrains from writing to the screen during the execution of a keyboard macro, or during typeahead. The command redisplay forces a complete rewrite of the screen. You may find this useful for writing macros that should update the screen in the middle of execution.

Summary:	Ctrl-X (	start-kbd-macro
	Ctrl-X)	end-kbd-macro
	Ctrl-F4, Ctrl-X E	last-kbd-macro
	Shift-F4	pause-macro
	Ctrl-X Alt-N	bind-last-macro
		name-kbd-macro
		insert-macro
		load-buffer
		redisplay

4.12 Simple Customizing

4.12.1 Bindings

Epsilon allows you to create your own commands and attach them, or any pre-existing Epsilon commands, to any key. If you bind a command to a key, you can then invoke that command by pressing the key. For example, at startup, Epsilon has forward-character bound to the Ctrl-F key. By typing Ctrl-F, the forward-character command executes, so point moves forward one character. If you prefer to have the command which moves point to the end of the current line, end-of-line, bound to Ctrl-F, you may bind that there.

You bind commands to keys with the bind-to-key command, which you can invoke with the F4 key. The bind-to-key command asks you for the name of a command (with completion), and the key to which to bind that command. You may precede the key by any number of *prefix keys*. When you type a prefix key, Epsilon asks you for another key. For example, if you type Ctrl-X, Epsilon asks you for another key. Suppose you type Ctrl-O. Epsilon would then bind the command to the Ctrl-X Ctrl-O key sequence. Prefix keys give Epsilon a virtually unlimited number of keys.

Epsilon at startup provides Ctrl-X and Ctrl-C as the only prefix keys. You can invoke many commands, such as save-file (Ctrl-X Ctrl-S) and find-file (Ctrl-X Ctrl-F), through the Ctrl-X prefix key. You may define your own prefix keys with the command called create-prefix-command. Epsilon asks you for a key to make into a prefix key. You may then bind commands to keys prefixed with this key using the bind-to-key command. To remove prefix keys, see page 133.

When you press a prefix key, Epsilon displays the key in the echo area to indicate that you must type another key. Epsilon normally displays the key immediately, but you can make it pause for a moment before displaying the key. If you press another key during the pause, Epsilon doesn't bother displaying the first key.

You control the amount of time Epsilon pauses using the `mention-delay` variable, expressed in tenths of a second. By default, this variable has a value of zero, which indicates no delay. You may find it useful to set `mention-delay` to a small value (perhaps 3). This delay applies in most situations where Epsilon prompts for a single key, such as when entering a numeric argument.

The unbind-key command asks for a key and then offers to rebind the key to the normal-character command, or to remove any binding it may have. A key bound to normal-character will self-insert; that's how keys like 'j' are bound. A key with no binding at all simply displays an error message.

You may bind a given command to any number of keys. You may invoke a command, whether or not bound to a key, using named-command, by pressing the Alt-X key. Alt-X asks for the name of a command, then runs the command you specified. This command passes any numeric argument you give it to the command it invokes.

The command alt-prefix, bound to ⟨Esc⟩, gets another key and executes the command bound to the Alt version of that key. You will find this command useful if you must use Epsilon from a keyboard lacking a working Alt key, or if you prefer to avoid using Alt keys. Also, you may find some combinations of control and alt awkward to type on some keyboards. For example, some people prefer to invoke the replace-string command by typing ⟨Esc⟩ & rather than by typing Alt-&.

The command ctrl-prefix, bound to Ctrl-^, functions similarly. It gets another key and converts it into the Control version of that key. For example, it changes 's' into the Ctrl-S key.

Some key combinations are variations of other key combinations. For instance, ⟨Backspace⟩ and Ctrl-H are related in this way. Epsilon uses a notion of generic versus specific keys; for instance, the specific key ⟨Backspace⟩ is also generically a Ctrl-H key. If you bind this key to a new command, Epsilon will ask if you want to bind only the ⟨Backspace⟩ key, or all key combinations that generate a Ctrl-H.

Summary: Alt-X, F2   named-command

F4	bind-to-key
	create-prefix-command
	unbind-key
⟨Esc⟩	alt-prefix
Ctrl-ˆ	ctrl-prefix

4.12.2 Brief Emulation

Epsilon can emulate the Brief text editor. The brief-keyboard command loads a Brief-style keyboard map. To undo this change, you can use the epsilon-keyboard command, which restores the standard keyboard configuration. This command only modifies those key combinations that Brief uses. Other keys retain their Epsilon definition. The Brief key map appears in figure 4.9.

In this release, Epsilon doesn't emulate a few parts of Brief. The separate commands for toggling case folding and regular expressions are not present, but you can type Ctrl-C and Ctrl-T within any searching command to toggle those things. Regular expressions follow Epsilon's syntax, not Brief's. Brief's commands for loading and saving keyboard macro files aren't implemented, since Epsilon lets you have an unlimited number of macros loaded at once, not just one. Epsilon will beep if you press the key of an unimplemented Brief emulation command.

In Brief, the shifted arrow keys normally switch windows. But Epsilon adopts the Windows convention that shifted arrow keys select text. In Brief mode, the Alt-arrow keys on the separate cursor pad may be used to switch windows.

You can make Epsilon's display resemble Brief's display using the set-display-look command. See page 93.

4.12.3 CUA Keyboard

In CUA emulation mode, Epsilon recognizes most of the key combinations commonly used in Windows programs. Other keys generally retain their usual Epsilon function.

To enable this emulation, press Alt-x, then type cua-keyboard and press ⟨Enter⟩. Use Alt-x epsilon-keyboard ⟨Enter⟩ to return to Epsilon's default key assignments.

The table shows the CUA key combinations that differ from Epsilon's native (Emacs-style) key configuration. In addition, various Alt-letter key combinations not mentioned here invoke menu items (for example, Alt-F displays the File menu in CUA mode, though it doesn't in Epsilon's native configuration).

Many commands in Epsilon are two-key combinations starting with Ctrl-X or Ctrl-C. In CUA mode, use Ctrl-W instead of Ctrl-X, and Ctrl-K instead of Ctrl-C. For example, the command delete-blank-lines, normally on Ctrl-X Ctrl-O, is on Ctrl-W Ctrl-O in CUA emulation.

4.12.4 Variables

You can set any user variable with the set-variable command. The variable must have the type *char*, *short*, *int*, *array of chars*, or *pointer to char*. The command first asks you for the name of the variable to set. You can use completion. After you select the variable, the command asks you for the new value. Then the command shows you the new value.

Whenever Epsilon asks you for a number, as in the set-variable command, it normally interprets the number you give in base 10. But you can enter a number in hexadecimal (base 16) by beginning the number with "0x", just like EEL integer constants. The prefix "0o" means octal, and "0b" means binary. For

Alt-a	mark-normal-region	Alt-1	brief-drop-bookmark 1
Alt-b	bufed	Alt-2	brief-drop-bookmark 2
Ctrl-B	line-to-bottom	...	...
Ctrl-C	center-window	Alt-0	brief-drop-bookmark 10
Alt-c	mark-rectangle	F1	move-to-window
Alt-d	kill-current-line	Alt-F1	toggle-borders
Ctrl-D	scroll-down	F2	brief-resize-window
Alt-e	find-file	Alt-F2	zoom-window
Ctrl-E	scroll-up	F3	brief-split-window
Alt-f	display-buffer-info	F4	brief-delete-window
Alt-g	goto-line	F5	string-search
Alt-h	help	Shift-F5	search-again
Alt-i	overwrite-mode	Alt-F5	reverse-string-search
Alt-j	brief-jump-to-	F6	query-replace
	bookmark	Shift-F6	replace-again
Ctrl-K	kill-line	Alt-F6	reverse-replace
Alt-k	kill-to-end-of-line	F7	record-kbd-macro
Alt-l	mark-line-region	Shift-F7	pause-macro
Alt-m	mark-inclusive-region	F8	last-kbd-macro
Ctrl-N	next-error	F10	named-command
Alt-n	next-buffer	Alt-F10	compile-buffer
Alt-o	set-file-name	Ctrl-⟨Enter⟩	brief-open-line
Alt-p	print-region	⟨Esc⟩	abort
Ctrl-P	view-process	⟨Del⟩	brief-delete-region
Alt-q	quoted-insert	⟨End⟩	brief-end-key
Alt-r	insert-file	⟨Home⟩	brief-home-key
Ctrl-R	argument	⟨Ins⟩	yank
Alt-s	string-search	Ctrl-⟨End⟩	end-of-window
Ctrl-T	line-to-top	Ctrl-⟨Home⟩	beginning-of-window
Alt-t	replace-string	Ctrl-⟨PgDn⟩	goto-end
Ctrl-U	redo-by-commands	Ctrl-⟨PgUp⟩	goto-beginning
Alt-u	undo-by-commands	Alt-⟨Minus⟩	previous-buffer
Alt-v	show-version	Ctrl-⟨Minus⟩	kill-buffer
Alt-w	save-file	Ctrl-⟨Bksp⟩	backward-kill-word
Ctrl-W	set-want-backup-file	Shift-⟨Home⟩	to-left-edge
Alt-x	exit	Shift-⟨End⟩	to-right-edge
Ctrl-X	write-files-and-exit	Num +	brief-copy-region
Alt-z	push	Num –	brief-cut-region
Ctrl-Z	zoom-window	Num *	undo-by-commands

Figure 4.9: Epsilon's key map for Brief emulation.

CUA Binding	Epsilon Binding	Command Name
Ctrl-A	Ctrl-X H	mark-whole-buffer
Ctrl-C	Alt-W	copy-region
Ctrl-F	Ctrl-S	incremental-search
Ctrl-H	Alt-R	query-replace
Ctrl-K ...	Ctrl-C ...	(prefix key: see below)
Ctrl-N		new-file
Ctrl-O	Ctrl-X Ctrl-F	find-file
Ctrl-P	Alt-F9	print-buffer
Ctrl-V	Ctrl-Y	yank ("paste")
Ctrl-W ...	Ctrl-X ...	(prefix key: see below)
Ctrl-X	Ctrl-W	kill-region ("cut")
Ctrl-Z	F9	undo
Alt-A	Ctrl-Z	scroll-up
Alt-Z	Alt-Z	scroll-down
Alt-O	Ctrl-X H	mark-paragraph
⟨Escape⟩	Ctrl-G	abort
F3	Ctrl-S Ctrl-S	search-again
⟨Home⟩	Ctrl-A	beginning-of-line
⟨End⟩	Ctrl-E	end-of-line

Figure 4.10: CUA Key Assignments

example, the numbers "30", "0x1E", "0o36", and "0b11110" all refer to the same number, thirty. You can also specify an ASCII value by enclosing the character in single quotes. For example, you could type 'a' to specify the ASCII value of the character "a" (in this example, 97).

The set-any-variable command is similar to set-variable, but also includes *system variables*. Epsilon uses system variables to implement its commands; unless you're writing EEL extensions, there's generally no reason to set them. When an EEL program defines a new variable, Epsilon considers it a system variable unless the definition includes the user keyword.

The show-variable command prompts for the name of the variable you want to see, then displays its value in the echo area. The same restrictions on variable types apply here as to set-variable. The command includes both user and system variables when it completes on variable names.

The edit-variables command in the non-GUI versions of Epsilon lets you browse a list of all variables, showing the current setting of each variable and the help text describing it, as you move through the list. You can use the arrow keys or the normal movement keys to move around the list, or begin typing a variable name to have Epsilon jump to that portion of the list. Press ⟨Enter⟩ to set the value of the currently highlighted variable, then edit the value shown using normal Epsilon commands. To exit from edit-variables, press ⟨Esc⟩ or Ctrl-G. With a numeric argument, the command includes system variables in its list.

In Epsilon for Windows, the edit-variables command behaves differently. It uses the help system to display a list of variables. After selecting a variable, press the Set button to alter its value.

Some Epsilon variables have a different value in each buffer. These *buffer-specific* variables take on a potentially different value each time the current buffer changes. Each buffer-specific variable also has a default value. Whenever you create a new buffer, you also automatically create a new copy of the buffer-specific variable as well. The value of this buffer-specific variable is initially this default value. In Epsilon's EEL extension language, you can define a buffer-specific variable by using the buffer storage

class specifier, and give it a default value by initializing it like a regular variable.

Just as Epsilon provides buffer-specific variables, it also provides *window-specific* variables. These have a different value for each window. Whenever you create a new window, you automatically create a new copy of the window-specific variable as well. When you split a window in two, both windows initially have the same values for all their window-specific variables. Each window-specific variable also has a default value. Epsilon uses the default value of a window-specific variable when it creates its first tiled window while starting up, and when it creates pop-up windows. You define a window-specific variable in EEL with the window storage class specifier, and you may give it a default value by initializing it like a regular variable.

If you ask the set-variable command to set a buffer-specific or window-specific variable, it will ask you if you mean the value for the current buffer (or window), or the default value, or both. You can also tell the set-variable command which value(s) you want to set by giving the command a numeric argument. Zero means set only the current value; any positive numeric argument means set both the current and default values; and any negative numeric argument means set only the default value.

Variables retain their values until you exit Epsilon, unless you make the change permanent with the write-state command, described on page 130. This command saves only the default value for buffer-specific and window-specific variables. It does not save the instantiated values of the variable for each buffer or window, since the buffers and windows themselves aren't listed in a state file. Session files, which do list individual buffers and windows, also record selected buffer-specific and window-specific variables.

The show-variable command will generally show you both the default and current values of a buffer-specific or window-specific variable. For string variables, though, the command will ask which you want to see.

The create-variable command lets you define a new variable without using the extension language. It asks for the name, the type, and the initial value.

You can delete a variable, command, macro, subroutine, or color scheme with the delete-name command, or rename one with the change-name command. Neither of these commands will affect any command or subroutine in use at the time you try to alter it.

Summary: F8   set-variable
                  Ctrl-F8   show-variable
                                                  set-any-variable
                                                  edit-variables
                                                  create-variable
                                                  delete-name
                                                  change-name

4.12.5   Saving Changes to Bindings and Variables

Epsilon can save any new bindings you have made and any macros you have defined for future editing sessions. Epsilon uses two kinds of files for this purpose, state files and command files (such as the einit.ecm file Epsilon normally uses to save your customizations).

The next section describes command files such as the einit.ecm file. For most users, the einit.ecm file is the best way to save customizations, but users with a large number of customizations may want to take advantage of the improved startup speed possible by using a state file instead.

Both methods can save bindings, macros, and other sorts of customizations, but they differ in many respects:

- A state file contains commands, macros, variables, and bindings. A command file can contain macros, many types of variables, and bindings, but it can't contain commands written in Epsilon's extension language. (It can contain requests to load other files containing such commands, though.)

- When Epsilon writes a state file, all currently defined commands, macros and variables go into it. A command file contains just what you put there.

- Epsilon can only read a state file during startup. It makes the new invocation of Epsilon have the same commands as the Epsilon that performed the write-state command that created that state file. By contrast, Epsilon can load a command file at any time.

- A command file appears in a human-readable format, so you can edit it as a normal file. By contrast, Epsilon stores a state file in a binary format. To modify a state file, you read it into a fresh Epsilon, use appropriate Epsilon commands (like bind-to-key to change bindings), then save the state with the write-state command.

- Epsilon can read a state file much faster than a command file.

- Binary state files from one release of Epsilon usually aren't compatible with state files from a different major release. Command files are.

The write-state command on Ctrl-F3 asks for the name of a file, and writes the current state to that file. The file name has its extension changed to ".sta" first, to indicate a state file. If you don't provide a name, Epsilon uses the name "epsilon.sta", the same name that it looks for at startup. (Under Unix, a state file name includes its major Epsilon version number.) You can specify another state file for Epsilon to use at startup with the −s flag.

By default, when you write a new state file, Epsilon makes a copy of the old one in a file named ebackup.sta. You can turn backups off by setting the variable `want-state-file-backups` to 0, or change the backup file name by modifying the `state-file-backup-name` template. See page 99 for information on templates.

Summary: Ctrl-F3   write-state

4.12.6 Command Files

A *command file* specifies macro definitions, key bindings, and other settings in a human-readable format, as described in the next section.

One important example of a command file is the `einit.ecm` file. Epsilon automatically loads this file each time you run it.

The −noinit flag tells Epsilon not to load an einit.ecm file. You can also set the `load-customizations` variable (and save the setting in your state file) to turn off reading an einit.ecm file.

You can use the list-customizations command to add a list of all the customizations in the current session of Epsilon to the end of this file.

Or you can set Epsilon to automatically record many types of customizations in this file. Set the `record-customizations` variable to 1 to tell Epsilon to record all such customizations, but not to automatically save them. Set it to 2 to record and save them without prompting.

With either method, you can always edit the customizations file to remove any settings you don't want, or use commands like insert-macro to add specific customizations to it.

Epsilon creates its einit.ecm file in your customization directory. Under Unix, this is `~/.epsilon`. Under Windows, its location depends on the version of Windows, but is often `\Documents and Settings\`*username*`\Application Data\Lugaru\Epsilon`. See page 11 for details. Epsilon searches for an einit.ecm file using your EPSPATH. The edit-customizations command is a convenient way to locate and start editing this file.

If you prefer to write customizations in EEL format, you can create an EEL source file named einit.e in the same directory as your einit.ecm file, and tell Epsilon to load it at startup by adding this line to your einit.ecm file:

```
(load-eel-from-path "einit.e" 2)
```

You can use the load-file command to make Epsilon load the settings in any command file. It asks you for the name of a file, then executes the commands contained in it. The load-buffer command asks you for the name of a buffer, then executes the commands contained in that buffer.

Set the variable `einit-file-name` to make Epsilon look for a file with a name other than einit.ecm. For instance, by using the command line flag `-d` to set this variable, you can make a particular invocation of Epsilon load a specialized set of commands and settings to carry out a noninteractive batch editing task.

Summary: edit-customizations
                                                                    load-file
                                                                    load-buffer

Command File Syntax

Epsilon's command files appear in a human-readable format, so you can easily modify them. Parentheses surround each command. Inside the parentheses appear a command name, and optionally one or more arguments. The command can be one of several special commands described in the next section, or most any EEL subroutine. See the next section for details.

Each argument can be either a number, a string, or a key list (a special type of string). Spaces separate one argument from the next. Thus, each command looks something like this:

```
(command-name "first-string" "second-string")
```

You can include comments in a command file by putting a semicolon or hash sign ('#') anywhere an opening parenthesis may appear. Such a comment extends to the end of the line. You cannot put a comment inside a string.

For numbers, you can include bases using a prefix of "0x" for hexadecimal, "0o" for octal, or "0b" for binary, or use an EEL-style character constant like `'X'` or `'\n'`. For strings, quote each `"` or `\` character with a `\`, as in EEL or C.

A few commands such as `define-macro` take a list of one or more keys; these use the syntax of strings, but with some additional rules. Most characters represent themselves, control characters have a "C-" before them, alt characters have an "A-", and function keys have an "F-" followed by the number of the function key. Cursor keys appear in a notation like `<Home>`. See page 139 for details.

Put a `\` before any of these sequences to quote them. For instance, if a macro should contain a Ctrl-F key, write `"C-F"`. If a macro should contain the three characters C hyphen F, write `"\C-F"`. The characters you have to quote are `<`, `"`, and `\`, plus some *letter-* sequences. Thus, the DOS file name `\job\letter.txt` in a string looks like `\\job\\letter.txt`. Do not put extra spaces in command file strings that represent keys. For example, the string `"C-X F"` represents "C-X `<Space>` F", not "C-X F" with no ⟨Space⟩.

You can also use the special key syntax <!*cmdname*> in a keyboard macro to run a command *cmdname* without knowing which key it's bound to. For example, <!find-file> runs the find-file command. When you define a keyboard macro interactively and invoke commands from the menu bar or tool bar, Epsilon will use this syntax to define them, since there may be no key sequence that invokes the specified command.

In addition to the above command syntax with commands inside parentheses, command files may contain lines that define variables, macros, key tables or bindings. Epsilon understands all the different types of lines generated by the list-all, list-customizations, import-customizations, and similar commands. When Epsilon records customizations in your `einit.ecm` file, it uses this line-by-line syntax for many types of customizations.

Besides listing variables, macros, key tables, and bindings, the above commands also create lines that report that a particular command or subroutine written in Epsilon's EEL extension language exists. These lines give the name, but not the definition, because command files can't define EEL functions. When Epsilon sees a line like that, it makes sure that a command or subroutine with the given name exists. If not, it reports an error. Epsilon does the same thing with variables that have complicated types (pointers or structures, for example).

Command File Examples

One common command in command files is *bind-to-key*. For bind-to-key, the first string specifies the name of some Epsilon command, and the second string represents the key whose binding you wish to modify, in a format we'll describe in detail in a moment. For instance, the following command binds the command show-matching-delimiter to }:

```
;   This example binds show-matching-delimiter to the
;   } character so that typing a } shows the matching
;   { character.
(bind-to-key "show-matching-delimiter" "}")
```

Unlike the regular command version, bind-to-key in a command file can unbind a prefix key. Say you want to make Ctrl-X no longer function as a prefix key, but instead have it invoke down-line. If, from the keyboard, you typed F4 to invoke bind-to-key, supplied the command name down-line, and then typed Ctrl-X as the key to rebind, Epsilon would assume you meant to rebind some *subcommand* of Ctrl-X, and wait for you to type a Ctrl-K, for instance, to bind down-line to Ctrl-X Ctrl-K. Epsilon doesn't know you have finished typing the key sequence. But in a command file, quotes surround each of the arguments to bind-to-key. Because of this, Epsilon can tell exactly where a key sequence ends, and you could rebind Ctrl-X as above (discarding the bindings available through Ctrl-X in the process) by saying:

```
(bind-to-key "down-line" "C-X")
```

In a command file, *define-macro* allows you to define a keyboard macro. Its first string specifies the name of the new Epsilon command to define, and its second string specifies the sequence of keys you want the command to type. The define-macro command does not correspond to any single regular Epsilon command, but functions like a combination of start-kbd-macro, end-kbd-macro, and name-kbd-macro.

The command file command *create-prefix-command* takes a single string, which specifies a key, and makes that key a prefix character. It works just as the regular command of the same name does.

Using the same syntax, you can also call EEL commands and subroutines. For subroutines, any that take only numeric and string arguments should work. Here are some useful ones:

```
(load-eel-from-path "file.e" 2)
(load-from-path "file.b")
```

The first line makes Epsilon search the EPSPATH for an EEL extension language source file named file.e, then compile and load it. (You can use an absolute path instead, remembering to double any backslashes.) Its second argument 2 makes the `load-eel-from-path` subroutine stop if there's an error; the value 1 won't complain if the file wasn't found but complains on other errors, and the value 0 suppresses all errors. The second line makes Epsilon search the EPSPATH for the compiled version of that same file and load it; this is faster, but it requires you to manually recompile that file when updating.

Many commands that prompt for some information like a file number aren't directly suitable for including in a command file; they don't know how to accept an argument supplied by a command file and will always prompt. In many cases an EEL subroutine is available that performs a similar task but is suitable for use in a command file.

Here's an example command file:

```
; This macro makes the window below the
; current one advance to the next page.
(define-macro "scroll-next-window" "C-XnC-VC-Xp")
(bind-to-key "scroll-next-window" "C-A-v")

;This macro asks for a file and puts
;it in another window.
(define-macro "split-and-find" "A-Xsplit-window
A-Xredisplay
A-Xfind-file
")
```

The first two lines contain comments. The third line begins the definition of a macro called scroll-next-window. It contains three commands. First Ctrl-x n invokes **next-window**, to move to the next window on the screen. The Ctrl-v key runs **next-page**, scrolling that window forward, and Ctrl-x p then invokes **previous-window**, to return to the original window. The fourth line of this example binds this new macro to the Ctrl-Alt-v key, so that from then on, typing a 'v' with Control and Alt depressed will scroll the next window forward.

The file defines a second macro named split-and-find. It invokes three commands by name. Notice that the macro could have invoked two of the commands by key. Invoking by name makes the macro easier to read and modify later. The redisplay command shows the action of split-window before the find-file command prompts the user for a file name.

Building Command Files

Rather than preparing command files according to the rules presented here, you may wish to have Epsilon write parts of them automatically. You can set the `record-customizations` variable so Epsilon does this automatically whenever you define a macro or bind a key, writing the definition to your `einit.ecm` file.

Or you can have Epsilon list all your current customization settings in a file. The list-customizations command inserts a list of all your customizations into the customization buffer, which contains your `einit.ecm` file. This includes bindings, macros, color settings, and other changes. The list-all command is similar, but lists every setting, even if it's not a customization (such as variable settings you've never customized).

Epsilon also has commands to create individual bind-to-key and define-macro lines needed to define a specific current binding or macro.

The insert-binding command asks you for a key, and inserts a bind-to-key command into the current buffer. When you load the command buffer, Epsilon will restore the binding of that key.

The insert-macro command creates an appropriate define-macro command for a macro whose name you specify, and inserts the command it builds into the current buffer. This comes in handy for editing an existing keyboard macro. You can use the edit-customizations command first to switch to the customization buffer, creating it if necessary.

Summary: insert-binding
                                                           insert-macro
                                                           list-customizations

4.12.7 Using National Characters

This section explains how to use Epsilon to edit text containing non-English characters such as ê or å.

Epsilon supports Unicode, as well as many 8-bit national character sets such as ISO 8859-1 (Latin 1).

Under Unix, full Unicode support is only available when Epsilon runs under X, and when a font using the iso10646 character set is in use. See http://www.lugaru.com/links.html#unicode for Unicode font sources.

Under Windows, full Unicode support is only available under Windows NT/2000/XP. For Unicode support in the Win32 Console version, see the `console-ansi-font` variable. Also see page 100 for more information on the DOS/OEM encoding used by default in the Win32 console version.

In this release, Epsilon doesn't display Unicode characters outside the basic multilingual plane (BMP), or include any of the special processing needed to handle complex scripts, such as scripts written right-to-left.

Epsilon knows how to translate between its native Unicode format and dozens of encodings and character sets (such as UTF-8, ISO-8859-4, or KOI-8).

Epsilon autodetects the encoding for files that start with a Unicode signature ("byte order mark"), and for many files that use the UTF-8 encoding. To force translation from a particular encoding, provide a numeric argument to a file reading command like find-file. Epsilon will then prompt for the name of the encoding to use. Press "?" when prompted for an encoding to see a list of available encodings. The special encoding "raw" reads and writes 8-bit data without any character set translation.

Epsilon uses the buffer's current encoding when writing or rereading a file. Use the set-encoding command to set the buffer's encoding.

The unicode-convert-from-encoding command makes Epsilon translate an 8-bit buffer in a certain encoding to its 16-bit Unicode version. The unicode-convert-to-encoding command does the reverse.

You can add a large set of additional converters to Epsilon by downloading a file. Mostly these converters add support for various Far East languages and for EBCDIC conversions. See http://www.lugaru.com/encodings.html for details.

Internally, buffers with no character codes outside the range 0–255 are stored with 8 bits per character; other buffers are stored with 16 bits per character. Epsilon automatically converts formats as needed.

The `detect-encodings` variable controls whether Epsilon tries to autodetect certain UTF-8 and UTF-16 files. The `default-read-encoding` variable says which encoding to use when autodetecting doesn't select an encoding. The `default-write-encoding` variable sets which encoding Epsilon uses to save a file with 16-bit characters and no specified encoding, in a context where prompting wouldn't be appropriate such as when auto-saving.

4.13   Advanced Topics

4.13.1   Changing Commands with EEL

Epsilon has many built-in commands, but you may want to add new commands, or modify the way some commands work. We used a language called EEL to write all of Epsilon's commands. You can find the EEL definitions to all of Epsilon's commands in files ending in ".e". EEL stands for Epsilon Extension Language.

Before you can load a group of commands from a ".e" file into Epsilon, you must compile them with the EEL compiler. You do this (outside of Epsilon, or in Epsilon's concurrent process buffer) by giving the command "eel *filename*" where *filename* specifies the name of the ".e" file you wish to compile (with or without the ".e"). The EEL compiler will read the source file and, if it finds no errors, will produce a "bytecode" file with the same first name but with a ".b" extension. A bytecode file contains command, subroutine, and variable definitions from the source file translated to a binary form that Epsilon can understand. It's similar to a regular compiler's object file.

Once you've compiled the file, the Epsilon load-bytes command gets it into Epsilon. This command prompts for a file name, then loads it into Epsilon. You may omit the extension. But an easier method is to load the EEL source file, then compile and load it in one step using the compile-buffer command on Alt-F3. See page 123.

Often a new EEL command won't work the first time. Epsilon incorporates a simple debugger to help you trace through the execution of a command. It provides single-stepping by source line, and you can enter a recursive edit level to locate point, display the values of global variables, or run test functions. The debugger takes the following commands:

⟨**Space**⟩ Step to the next line. This command will trace a function call only if you have enabled debugging for that function.

S If the current line calls a function, step to its first line. Otherwise, step to the current function's next line.

G Cancel debugging for the rest of this function call and let the function run. Resume debugging if someone calls the current function again.

R Begin a recursive edit of the current buffer. You may execute any command, including show-variable or set-variable. Ctrl-X Ctrl-Z resumes debugging the stopped function. (When debugging a function doing input, you may need to type Ctrl-U Ctrl-X Ctrl-Z to resume debugging.)

T Toggle whether or not the current function should start the debugger when called the next time. Parentheses appear around the word "Debug" in the debug status line to indicate that you have not enabled debugging for the current function.

+ Enlarge the debug window.

– Shrink the debug window.

? List all debugger commands.

To start the debugger, use the set-debug command. It asks for the name of a command or subroutine, providing completion, and toggles debugging for that function. (A zero numeric argument turns off debugging for that function. A nonzero numeric argument turns it on. Otherwise, it toggles.)

Compiling a file with the −s EEL compiler flag disables debugging for routines defined in that file. See page 155 for information about the EEL command line options, including the −s flag.

The profile command shows where a command spends its time. When you invoke the profile command, it starts a recursive edit level, and collects timing information. Many times each second, Epsilon notes the

source file and source line of the EEL code then executing. When you exit from the recursive edit with Ctrl-X Ctrl-Z, Epsilon displays the information to you in a buffer.

Epsilon doesn't collect any profiling information on commands or subroutines that you compile with the −s EEL flag.

The list-undefined command makes a list of EEL functions that are called from some other EEL function, but have no definition. These are typically the result of misspelled function names.

Summary: load-bytes
                                            set-debug
                                            profile
                                            list-undefined

4.13.2   Updating from an Old Version

When you update to a new release of Epsilon, you'll probably want to incorporate any customizations you've made into the new version. You can save customizations in two ways: in a customization file called einit.ecm (see page 131) (a simple text file which this version and future versions of Epsilon can read), or in a state file (a binary file that's specific to a particular major release). Prior to version 12, Epsilon only supported the latter method.

If you're updating from the Windows or Unix versions of Epsilon version 8 or later, use the import-customizations command to run the previous version and collect its customizations, inserting them into your einit.ecm file for use in Epsilon 12 and future versions.

Once you've done that, future versions will automatically use your customizations. If you make more customizations, and choose to save them in a state file, not an einit.ecm file, you can use the list-customizations command to copy your current customizations to an einit.ecm file.

The next section explains how to update from an older version where import-customizations isn't supported.

One exception to the above is if you've customized Epsilon by writing Epsilon extensions in Epsilon's extension language, EEL. The list-customizations and import-customizations commands will help you insert references to your EEL source code files into your einit.ecm file, but it's possible that some of the built-in functions or subroutines called by your EEL source code files have changed. In that case, you will have to modify your commands to take this into account. See the section on changes from previous versions in the online manual for information on EEL changes.

Updating from Epsilon 7.0 or Older Versions

This section explains how to transfer customizations from a version of Epsilon so old that the import-customizations command isn't supported. The import-customizations command is available when updating from version 8 or later, under Windows or Unix.

This section also applies if you're updating from a DOS or OS/2 implementation of Epsilon, regardless of its version, since import-customizations is only supported for the Windows and Unix implementations.

Moving your customizations (such as variable settings, changed bindings, or keyboard macros) from your old version of Epsilon into the new version requires several steps.

• Start the old version of Epsilon as you do normally.

- Run the list-all command.

  This will make a list of all the variables, bindings, macros, and functions defined in your old version of Epsilon.

- Save the result in a file. We will assume you wrote it to a file named "after".

- You should no longer need the old version of Epsilon, so you can now install the new version in place of the old one if you wish. Or you can install the new version in a separate directory.

- Locate the "changes" subdirectory within Epsilon's main directory.

For each old version of Epsilon, you'll need several files in the steps below. In the description that follows, we will assume that you want to move from Epsilon 7.0 to this version, and will use files with names like list70.std. Substitute the correct file name if you have a different version (for example, list40.std to upgrade from Epsilon 4).

- Locate the file in the changes subdirectory from the new version of Epsilon with a name like list70.std. It resembles the "after" file, but comes from an unmodified copy of that version of Epsilon. We will call this the "before" file. If you have a very old version for which there is no .std file, see page 139 to make one.

- Start the new version of Epsilon. Run the list-changes command. It will ask for the names of the "before" and "after" files, and will then make a list of differences between the files, a "changed" file. When it finishes, you will have a list of the changes you made to the old version of Epsilon, in the format used by the list-all command. Edit this to remove changes you don't want in the new version, and save it.

- Run the load-changes command, and give it the name of the "changed" file from the previous step. It will load the changes into Epsilon. You can define commands, subroutines, and some variables only from a compiled EEL file, not via load-changes. If any of these appear in your changed file, Epsilon will add a comment after that line, stating why it couldn't make the change.

- Use the list-customizations command to add your customizations to Epsilon's einit.ecm file.

Note that this procedure will not spot changes made in .e files, only those made to variables, bindings or macros. It will notice if you have defined a new command, but not if you have modified an existing command.

The above procedure uses several commands. The list-all command lists the current state of Epsilon in text form, mentioning all commands and subroutines, and describing all key bindings, macros, and variables. The list-changes command accepts the names of the "before" and "after" files produced by list-all, and runs the compare-sorted-windows command on them to make a list of the lines in "after" that don't match a line in "before".

Finally, the load-changes command reads this list of differences and makes each modification listed. It knows how to create variables, define macros, and make bindings, but it can't transfer extension-language commands. You'll have to use the new EEL compiler to incorporate any EEL extensions you wrote.

Updating from Epsilon 4.0 or Older Versions

If you're updating from a version of Epsilon before 4.0, you'll have to make several files before updating. You will need your old version of Epsilon (including the executable program files for Epsilon and EEL), the state file you've been using with it (typically named epsilon.sta), and the original state file that came with

that version of Epsilon (which you can find on your old Epsilon distribution disk). You'll also need the file list-all.e, included with the new version of Epsilon. First, read the comments in the file list-all.e and edit it as necessary to match your version. Then compile it with the old EEL compiler. This will create the bytecode file list*version*.b. Start your old version of Epsilon with its original state file, using a command like `epsilon -s\oldver\epsilon`, and load the bytecode file you just created, using the load-bytes command on the F3 key. Now save the resulting list in a file named "before". Then start your old version of Epsilon again, this time with your modified state file, and load the bytecode file list*version*.b again. Now save the resulting list in a file named "after". Next, start the new version of Epsilon, read in the "before" file, and sort using the sort-buffer command, and write it back to the "before" file. You can now continue with the procedure above, running the list-changes command and providing the two files you just created.

If we didn't provide a .std file for your version of Epsilon, and you're running Epsilon 4.0 or later, here's how to make one. You will need your old version of Epsilon, the state file you've been using with it (typically named epsilon.sta), and the original state file that came with that version of Epsilon (which you can find on your old Epsilon distribution disk). Start your old version of Epsilon with its original state file, using a command like `epsilon -s\oldver\epsilon`, and run the list-all command. Now save the resulting list in a file named "before". Then start your old version of Epsilon again (just as you normally do) using the state file that contains the changes you've made, and run the list-all command again. Now save the resulting list in a file named "after". Next, start the new version of Epsilon, read in the "before" file, and sort using the sort-buffer command, and write it back to the "before" file. You can now continue with the procedure above, running the list-changes command and providing the two files you just created.

Summary: list-all
                                                     list-changes
                                                     load-changes

4.13.3   Keys and their Representation

This section describes the legal Epsilon keys, and the representation that Epsilon uses when referring to keys and reading command files. The key representation used when writing extension language programs appears on page 304.

Epsilon recognizes hundreds of distinct key combinations you can type on the keyboard (including control and alt keys). You can bind a command to each of these keys. Each key can also function as a prefix key, allowing even more key combinations. By default, Ctrl-X and Ctrl-C serve as prefix keys.

First, the keyboard provides the standard 128 ASCII characters. All the white keys in the central part of the PC keyboard, possibly in combination with the Shift and Control keys, generate ASCII characters. So do the ⟨Esc⟩, ⟨Backspace⟩, ⟨Tab⟩, and ⟨Enter⟩ keys. They generate Control [, Control H, Control I, and Control M, respectively. Depending upon the national-language keyboard driver in use, there may be up to 128 additional keys available by pressing various combinations of Control and AltGr keys, for a total of 256 keys.

You can get an additional 256 keys by holding down the Alt key while typing the above keys. In Epsilon, you can also enter an Alt key by typing an ⟨Esc⟩ before the key. Similarly, the Control-ˆ key says to interpret the following key as if you had held down the Control key while typing that key.

If you want to enter an actual ⟨Esc⟩ or Control-ˆ instead, type a Control-Q before it. The Ctrl-Q key "quotes" the following key against special interpretations. See page 126.

In command files and some other contexts, Epsilon represents Control keys by C-⟨char⟩, with ⟨char⟩ replaced by the original key. Thus Control-t appears as C-T. The case of the ⟨char⟩ doesn't matter for control characters when Epsilon reads a command file, but the C- must appear in upper case. The Delete

```
<Ins>          <Insert>
<End>
<Down>
<PgDn>         <PageDn>      <PgDown>      <PageDown>
<Left>
<Right>
<Home>
<Up>
<PgUp>         <PageUp>
<Del>          <Delete>
```

Figure 4.11: Names Epsilon uses for the cursor keypad keys.

character (ASCII code 127) appears as C-?. Note that this has nothing to do with the key marked "Del" on the PC keyboard. The Alt keys appear with A- appended to the beginning of their usual symbol, as in A-f for Alt-f and A-C-h for Alt-Control-H.

Epsilon represents function keys by F-1, F-2, . . . F-12. The F must appear in upper case. You can also specify the Shift, Control, and Alt versions of the function keys. When typing a function key, Epsilon only recognizes one "modifier." If you press the Alt key down, Epsilon ignores the Control and Shift keys, and if you press the Control key down, Epsilon ignores the Shift key. You can specify 48 keys in this way. In a command file, you specify the Shift, Control, and Alt versions with a prefix of S-, C-, or A-, respectively. For example, Epsilon refers to the key you get by holding down the Shift and Alt keys and pressing the F8 key as A-F-8.

Keys on the cursor keypad work in a similar way. Epsilon recognizes several synonyms for these keys, as listed in figure 4.11. Epsilon generally uses the first name listed, but will accept any of the names from a command file. Epsilon recognizes Control, Shift, and Alt versions of these keys as well, and indicates these with a prefix of C-, S-, or A-, respectively. You can only use one of these prefixes at a time.

Epsilon normally treats the shifted versions of these keys as synonyms for the unshifted versions. When you press Shift-⟨Left⟩, Epsilon runs the command bound to ⟨Left⟩. The commands bound to most of these keys then examine the Shift key and decide whether to begin or stop selecting text. (Holding down the shift key while using the cursor keys is one way to select text in Epsilon.)

Epsilon refers to the numeric keypad keys with the names given in figure 4.12.

In a command file, you can also represent keys by their conventional names, by writing <Newline> or <Escape>, or by number, writing <#0> for the null character ^@, for example. Epsilon understands the same key names here as in regular expression patterns (see figure 4.3 on page 57).

Macros defined in command files may also use the syntax <!*cmdname*> to run a command *cmdname* without knowing which key it's bound to. For example, <!find-file> runs the find-file command. When you define a keyboard macro interactively and invoke commands from the menu bar or tool bar, Epsilon will use this syntax to define them, since there may be no key sequence that invokes the specified command.

Several keys on the PC keyboard act as synonyms for other keys: the grey keys *, −, and + by the numeric keypad, and the ⟨Backspace⟩, ⟨Enter⟩, ⟨Tab⟩, and ⟨Esc⟩ keys, for example. The first three act as synonyms for the regular white ASCII keys, and the other four act as synonyms for the Control versions of 'H', 'M', 'I' and '[', respectively. Epsilon normally translates these keys to their synonyms automatically, and uses the binding of the synonym, but you can also bind them separately if you prefer, using the specific key names shown in figure 4.13.

```
N-<Ins>        N-<Insert>     N-0
N-<End>        N-1
N-<Down>       N-2
N-<PgDn>       N-<PageDn>     N-<PgDown>     N-<PageDown>  N-3
N-<Left>       N-4
N-5
N-<Right>      N-6
N-<Home>       N-7
N-<Up>         N-8
N-<PgUp>       N-<PageUp>     N-9
N-<Del>        N-<Delete>     N-.
```

Figure 4.12: Numeric keypad key names recognized and displayed by Epsilon.

Specific Key	Becomes Generic Key	
⟨NumPlus⟩	+	
⟨NumMinus⟩	−	
⟨NumStar⟩	*	
⟨NumSlash⟩	/	
⟨EnterKey⟩	⟨Enter⟩	(on main keyboard)
⟨NumEnter⟩	⟨Enter⟩	(on numeric keypad)
⟨BackspaceKey⟩	⟨Backspace⟩	
⟨TabKey⟩	⟨Tab⟩	
⟨EscapeKey⟩	⟨Esc⟩	
⟨Spacebar⟩	⟨Space⟩	

Figure 4.13: Some keys that are synonyms for others.

Mouse Keys

When you use the mouse, Epsilon generates a special key code for each mouse event and handles it the same way as any other key. (For mouse events, Epsilon also sets certain variables that indicate the position of the mouse on the screen, among other things. See page 306.)

```
M-<Left>      M-<LeftUp>     M-<DLeft>      M-<Move>
M-<Center>    M-<CenterUp>   M-<DCenter>
M-<Right>     M-<RightUp>    M-<DRight>
```

Epsilon uses the above names for mouse keys when it displays key names in help messages and similar contexts. M-<Left> indicates a click of the left button, M-<LeftUp> indicates a release, and M-<DLeft> a double-click. See page 142 before binding new commands to these keys.

Epsilon doesn't record mouse keys in keyboard macros. Use the equivalent keyboard commands when defining a macro.

There are several "input events" that Epsilon records as special key codes. Their names are listed below. See page 312 for information on the meaning of each key code.

```
M-<MenuSel>   M-<HScroll>    M-<WinHelpReq>   M-<LoseFocus>
M-<Resize>    M-<DragDrop>   M-<Button>
M-<VScroll>   M-<WinExit>    M-<GetFocus>
```

Under Windows, Epsilon displays a tool bar. The toggle-toolbar command hides or displays the tool bar. To modify the contents of the tool bar, see the definition of the standard-toolbar command in the file menu.e, and the description of the tool bar primitive functions starting on page 276.

The invoke-windows-menu command brings up the Windows system menu. Alt-⟨Space⟩ is bound to this command. If you bind this command to an alphabetic key like Alt-P, it will bring up the corresponding menu (the Process menu, in this example).

In a typical Windows program, pressing and releasing the Alt key without pressing any other key moves to the menu bar, highlighting its first entry. Set the variable `alt-invokes-menu` to one if you want Epsilon to do this. The variable has no effect on what happens when you press Alt and then press another key before releasing Alt: this will run whatever command is bound to that key. If you want Alt-E, for example, to display the Edit menu, you can bind the command invoke-windows-menu to it.

Summary: toggle-toolbar
                                                  invoke-windows-menu

4.13.4   Customizing the Mouse

You can rebind the mouse buttons in the same way as other keys using the bind-to-key command, but if, for example, you rebind the left mouse button to copy-region, then that button will copy the region from point to mark, regardless of the location of the mouse. Instead, you might want to use the left button to select a region, and then copy that region. To do this, leave the binding of the left mouse button alone, and instead define a new version of the mouse-left-hook function. By default, this is a subroutine that does nothing. You can redefine it as a keyboard macro using the name-kbd-macro command. Epsilon runs this hook function after you release the left mouse button, if you've used the mouse to select text or position point (but not if, for example, you've clicked on the scroll bar).

Normally Epsilon runs the mouse-select command when you click or double-click the left mouse button, and the mouse-to-tag command when you click or double-click the right mouse button. Epsilon runs the mouse-move command when you move the mouse; this is how it changes the mouse cursor shape or pops up a scroll bar or menu bar when the mouse moves to an appropriate part of the screen, in some environments.

Both mouse-select and mouse-to-tag run the appropriate hook function for the mouse button that invoked them, whenever you use the mouse to select text or position point. The hook functions for the other two mouse buttons are named mouse-right-hook and mouse-center-hook. You can redefine these hooks to make the mouse buttons do additional things after you select text, without having to write new commands using the extension language. (Note that in Epsilon for Windows mouse-to-tag displays a context menu instead of selecting text, by calling the context-menu command, and doesn't call any hook function.)

By default, the center mouse button runs the command mouse-center, which calls either mouse-yank to make the mouse yank text from the clipboard under Unix, or mouse-pan, so that button makes the mouse scroll or pan (in other environments).

Summary: M-⟨Left⟩   mouse-select
                      M-⟨Right⟩   mouse-to-tag
                      M-⟨Center⟩   mouse-center

M-⟨Move⟩	mouse-move
	mouse-pan
	mouse-yank
	context-menu

4.14 Miscellaneous

You can use the eval command to quickly evaluate an arbitrary EEL expression, or do simple integer-only math. By default, the command displays the result; use a numeric prefix argument and it will insert the result in the current buffer. Similarly, the execute-eel command executes a line of EEL code that you type in.

The command narrow-to-region temporarily restricts your access to the current buffer to the region between the current values of point and mark. Epsilon hides the portion of the buffer outside this region. Searches will only operate in the narrowed region. While running with the buffer narrowed, Epsilon considers the buffer to start at the beginning of the region, and end at the end of the region. However, if you use a file-saving command with the buffer narrowed in this manner, Epsilon will write the entire file to disk. To restore normal access to the buffer, use the widen-buffer command.

Under Windows, you can set Epsilon's key repeat rate with the `key-repeat-rate` variable. It contains the number of repeats to perform in each second. Setting this variable to 0 lets the operating system or keyboard determine the repeat rate, as it does outside of Epsilon. Epsilon never lets repeated keys pile up; it ignores automatically repeated keys when necessary.

Summary:	narrow-to-region
	widen-buffer
	eval
	execute-eel

Chapter 5

Changing Epsilon

Epsilon provides several ways for you to change its behavior. Some commands enable you to make simple changes. For example, set-fill-column can change the width of filled lines of text. Commands like bind-to-key and create-prefix-command can move commands around on the keyboard, and using keyboard macros, you can build simple new commands. The remaining chapters of the manual describe how to use the Epsilon Extension Language, EEL, to make more sophisticated commands and to modify existing commands.

Unless you save them, all these types of changes go away when you exit, and you must reload them the next time you run Epsilon. (But see the record-customizations variable.)

There are many ways to save such changes. The easiest way to save them is with the list-customizations command. See page 131.

You can also save changes by storing them in a state file. This method is quicker if you have hundreds of customizations, or have made many customizations using the EEL extension language. This chapter describes various ways to do that.

When it starts, Epsilon reads a state file named epsilon.sta containing all of Epsilon's initial commands, variables, and bindings. (You can use Epsilon's -s flag to make Epsilon load its state from some other file. For example, "epsilon -sfilename" loads its commands from the file filename.sta.)

You can change Epsilon's set of commands and settings by generating a new state file with the Epsilon command write-state on Ctrl-F3. So one way to customize Epsilon is to make your changes (bind some keys, set a variable, define some macros) and use the write-state command to put the changes in epsilon.sta. Your customizations will take effect each time you run Epsilon. See page 130 for more on write-state.

Instead of manually setting variables, then saving them in a (binary) state file, you may want to preserve your changes in a human-readable format. Commands like list-customizations provide one way to do that, preserving variable settings and the like in Epsilon's command file format. You can also do this using EEL.

In that case, you may find it handy to have a file that loads your changes into a fresh Epsilon, then writes the new state file automatically. The following simple EEL file, which we'll call changes.e, uses features described in later chapters to do just that:

```
#include "eel.h"  /* Load standard definitions. */

when_loading()    /* Execute this file when loaded. */
{
    want_bell = 0;            /* Turn off the bell. */
    kill_buffers = 6;         /* Use only 6 kill buffers. */
    load_commands("mycmds");  /* Load my new commands. */
    do_save_state("epsilon"); /* Save these changes. */
}
```

Each time you get an update of Epsilon, you can compile this program (type eel changes outside of Epsilon) and start Epsilon with its new state file (type epsilon). Then when you load this file (type F3 changes ⟨Enter⟩ to Epsilon), Epsilon will make all your changes in the updated version and automatically save them for next time.

You can change most variables as in the example above. Some variables, however, have a separate value for each buffer. Consider, for example, the tab size (which corresponds to the value of the tab-size variable). This variable's value can potentially change from buffer to buffer. We call this a buffer-specific variable. Buffer-specific variables have one value for each buffer plus a special value called the default value. The default value specifies the value for the variable in a newly created buffer. A state file stores only the default value of a buffer-specific variable.

Thus, to change the tab size permanently, you must change tab_size's default value. You can use the set-variable command to make the change, or an EEL program. The following version of changes.e sets the default tab size to 5.

```
#include "eel.h"  /* load standard definitions */

when_loading()    /* execute this file when loaded */
{
    tab_size.default = 5;     /* set default value  */
    load_commands("mycmds");  /* load my new cmnds */
    do_save_state("epsilon"); /* save these changes */
}
```

Once you've learned a little EEL, you may want to modify some of Epsilon's built-in commands. We recommend that you keep your modifications to Epsilon in files other than the standard distributed source files. That way, when you get an update of Epsilon, you will find it easy to recompile your changes without accidentally loading in old versions of some of the standard functions.

You cannot redefine a function during that function's execution. Thus, changing the load-bytes command, for example, would seem to require writing a different command with the same functionality, and using each to load a new version of the other. You don't have to do this, however. Using the −b flag, you can load an entire system into Epsilon from bytecode files, not reading a state file at all. Epsilon does not execute any EEL functions while loading commands with the −b flag, so you can redefine any function using this technique.

To use this technique, first compile all the files that make up Epsilon. If you have a "make" utility program, you can use the makefile included with Epsilon to do this. Then start Epsilon with the −b flag. This loads the single bytecode file epsilon.b, which automatically loads all the others. The makefile then has Epsilon write a new state file using these definitions. If you have made extensive changes to Epsilon's commands, this method may be most convenient.

Chapter 6

Introduction to EEL

6.1   Epsilon Extension Language

The Epsilon Extension Language (EEL) allows you to write your own commands and greatly modify and customize the editor to suit your style. EEL provides a great deal of power. We used it to write all of Epsilon's commands. You can use it to write new commands, or to modify the ones that we provide.

We call EEL an *extension language* because you use it to extend the editor. Some people call such things *macro languages*. We use the term "macro" to refer to the keyboard macros you can create in Epsilon, or to EEL's C-like textual macros, but not to the commands or extensions you write in EEL.

EEL has quite a few features that most extension languages don't:

- Block structure, with a syntax resembling the **C programming language**.

- Full flow control: **if**, **while**, **for**, **do**, **switch** and **goto**. Additionally, EEL has a non-local goto facility provided by **setjmp** and **longjmp**.

- Complete set of data types, including **integers**, **arrays**, **structures**, and **pointers**. In addition, you may define new data types and allocate data objects dynamically.

- Subroutines with **parameter passing**. You may invoke subroutines **recursively**, and can designate any subroutine a command.

- Rich set of **arithmetic** and **logical** operators. EEL has all the operators of the C programming language.

- A powerful set of primitives. We wrote **all** of Epsilon's commands in EEL.

- Global variables accessible everywhere, and local variables accessible only in the current routine. EEL also has **buffer-specific variables** that change from buffer to buffer, and **window-specific variables** that have a different value in each window.

In addition, the runtime system provides a *source level tracing debugger*, and an *execution profiler*.

Epsilon's source subdirectory contains the EEL source code to all Epsilon's commands. You may find it helpful to look at this source code when learning the extension language. Even after you've become a proficient EEL programmer, you probably will find yourself referring to the source code when writing your own extensions, to see how a particular command accomplishes some task.

6.2   EEL Tutorial

This section will take you step by step through the process of creating a new command using EEL. You will learn how to use the EEL compiler, a few control structures and data types, and a few primitive operations. Most important, this section will teach you the mechanics of writing extensions in EEL.

As our example, we will write a simplified version of the insert-file command called simple-insert-file. It will ask for the name of a file, and insert the contents of the file before point in the current buffer. We will write it a few lines at a time, each time having the command do more until the whole command works. When you write EEL routines, you may find this the way to go. This method allows you to debug small sections of code.

Start Epsilon in a directory where you want to create the files for this tutorial. Using the find-file command (Ctrl-X Ctrl-F), create a file with the name "learn.e".

To write an extension, you: **write** the source code, **compile** the source code, **load** the compiled code, then **run** the command.

First, we write the source code. Type the following into the buffer and save it:

```
#include "eel.h"          /* standard definitions */

command simple_insert_file()
{
    char inserted_file[FNAMELEN];

    get_file(inserted_file, "Insert file", "");
    say("You typed file name %s", inserted_file);
}
```

Let's look at what the source code says. The first line includes the text of the file "eel.h" into this program, as though you had typed it yourself at that point.

Comments go between /* and */.

The file "eel.h" defines some system-wide constants, and a few global variables. Always include it at the beginning of your extension files.

The line

```
command simple_insert_file()
```

says to define a command with the name `simple_insert_file`. The empty parentheses mean that this function takes no parameters. The left brace on the next line and the right brace at the end of the file delimit the text of the command.

Each command or subroutine begins with a sequence of local variable declarations. Our command has one, the line

```
char inserted_file[FNAMELEN];
```

which declares an array of characters called `inserted_file`. The array has a length of FNAMELEN. The constant FNAMELEN (defined in eel.h) may vary from one operating system to another. It specifies the maximum file name length, including the directory name. The semicolon at the end of the line terminates the declaration.

The next statement

```
get_file(inserted_file, "Insert file", "");
```

calls the built-in subroutine `get_file()`. This primitive takes three parameters: a character array to store the user's typed-in file name, a string with which to prompt the user, and a value to offer as a default. In this case, the Epsilon will prompt the user with the text between the double quotes (with a colon stuck on the end). We call a sequence of characters between double quotes a *string constant*.

When the user invokes this command, the prompt string appears in the echo area. Epsilon then waits for the user to enter a string, which it copies to the character array. While typing in the file name, the user may use Epsilon's file name completion and querying facility. This routine returns when the user hits the ⟨Enter⟩ key.

The next statement,

```
say("You typed file name %s", inserted_file);
```

prints in the echo area what file name the user typed in. The primitive `say()` takes one or more arguments. The first argument acts as a template, specifying what to print out. The "%s" in the above format string says to interpret the next argument as a character array (or a string), and to print that instead of the "%s". In this case, for the second argument we provided `inserted_file`, which holds the name of the file obtained in the previous statement.

For example, say the user types the file name "foo.bar", followed by ⟨Enter⟩. The character array `inserted_file` would have the characters "foo.bar" in it when the `get_file()` primitive returns. Then the second statement would print out

```
You typed file name foo.bar
```

in the echo area.

One way to get this command into Epsilon is to run the EEL compiler to compile the source code into a form Epsilon can interpret, called a bytecode file. EEL source files end in ".e", and the compiler generates a file of compiled binary object code that ends in ".b". After you do that, you can load the .b file using the load-bytes command.

But an easier way that combines these steps is to use Epsilon's compile-buffer command on Alt-F3. This command invokes the EEL compiler, as if you typed

```
eel filename
```

where *filename* is the name of the file you want to compile, and then (if there are no errors) loads the resulting bytecode file. You should get the message "learn.b compiled and loaded." in the echo area.

Now that you've compiled and loaded learn.b, Epsilon knows about a command named simple-insert-file. Epsilon translates the underscores of command names to hyphens, so as to avoid conflicts with the arithmetic minus sign in the source text. So the name `simple_insert_file` in the eel source code defines simple-insert-file at command level.

Go ahead and invoke the command simple-insert-file. The prompt

```
Insert file:
```

appears in the echo area. Type in a file name now. You can use all Epsilon's completion and querying facilities. If you press '?', you will get a list of all the files. If you type "foo?", you will get a list of all the files that start with "foo". ⟨Esc⟩ and ⟨Space⟩ completion work. You can abort the command with Ctrl-G.

After you type a file name, this version of the command simply displays what you typed in the echo area.

Let's continue with the simple-insert-file command. We will create an empty temporary buffer, read the file into that buffer, transfer the characters to our buffer, then delete the temporary buffer. Also, let's get rid of the line that displays what you just typed. Make the file learn.e look like this:

```
#include "eel.h"          /* standard definitions */

command simple_insert_file()
{
        char inserted_file[FNAMELEN];
        char *original_buffer = bufname;

        get_file(inserted_file, "Insert file", "");
        zap("tempbuf");      /* make an empty buffer */
        bufname = "tempbuf"; /* use that buffer */
        if (file_read(inserted_file, 1) != 0)
                error("Read error: %s", inserted_file);
                        /* copy the characters */
        xfer(original_buffer, 0, size());
                        /* move back to buffer */
        bufname = original_buffer;
```

```
                delete_buffer("tempbuf");
}
```

This version has one more declaration at the beginning of the command, namely

```
char *original_buffer = bufname;
```

This declares `original_buffer` to point to a character array, and initializes it to point to the array named `bufname`.

The value of the variable `bufname` changes each time the current buffer changes. For this reason, we refer to such variables as *buffer-specific variables*. At any given time, `bufname` contains the name of the current buffer. So this initialization in effect stores the name of the current buffer in the local variable `original_buffer`.

After the `get_file()` call, we create a new empty buffer named "tempbuf" with the statement "`zap("tempbuf");`". We then make "tempbuf" the current buffer by setting the bufname variable with the following.

```
bufname = "tempbuf";
```

Now we can read the file in:

```
if (file_read(inserted_file, 1) > 0)
        error("Read error: %s", inserted_file);
```

This does several things. First, it calls the `file_read()` primitive, which reads a file into the current buffer. It returns 0 if everything goes ok. If the file doesn't exist, or some other error occurs, it returns a nonzero error code. The actual return value in that case indicates the specific problem. This statement, then, executes the line

```
error("Read error: %s", inserted_file);
```

if an error occurred while reading the file. Otherwise, we move on to the next statement. The primitive `error()` takes the same arguments that `say()` takes. It prints out the message in the echo area, aborts the command, and drops any characters you may have typed ahead.

Now we have the text of the file we want to insert in a buffer named `tempbuf`. The next statement,

```
xfer(original_buffer, 0, size());
```

calls the primitive `xfer()`, which transfers characters from one buffer to another. The first argument specifies the name of the buffer to transfer characters to. The second and third arguments give the region of the current buffer to transfer. In this case, we want to transfer characters to `original_buffer`, which holds the name of the buffer from which we invoked this command. We want to transfer the whole thing, so we give it the parameters 0 and `size()`. The primitive `size()` returns the number of characters in the current buffer.

The last two statements return us to our original buffer and delete the temporary buffer.

The final version of this command adds several more details.

On the first line, we've added on `cx_tab['i']`. This tells Epsilon to bind the command to Ctrl-X I. We've added a new character pointer variable named `buf`, because we will use Epsilon's `temp_buf()` subroutine for our temporary buffer rather than the wired-in name of "tempbuf". This subroutine makes up an unused buffer name and creates it for us. It returns the name of the buffer.

```
#include "eel.h"          /* standard definitions */

char region_file[FNAMELEN];

command simple_insert_file() on cx_tab['i']
{
    char inserted_file[FNAMELEN], *buf;
    char *original_buffer = bufname;
    int err;

    iter = 0;
    get_file(inserted_file, "Insert file", region_file);
    mark = point;
    bufname = buf = temp_buf();
    err = file_read(inserted_file, 1);
    if (!err)
        xfer(original_buffer, 0, size());
    bufname = original_buffer;
    delete_buffer(buf);
    if (err)
        file_error(err, inserted_file, "read error");
    else
        strcpy(region_file, inserted_file);
}
```

Figure 6.1: The final version of simple-insert-file

The line

```
    mark = point;
```

causes Epsilon to leave the region set around the inserted text. The `xfer()` will insert its text between mark and point. We've added the line `iter = 0;` to make the command ignore any numeric argument. Without this line, it would ask you for a file to insert over and over, if you accidentally gave it a numeric argument.

We now save the error code that `file_read()` returns so we can delete the temporary buffer in the event of an error. We also use the `file_error()` primitive rather than `error()` because the former will translate system error codes to text.

Finally, we added the line

```
    char region_file[FNAMELEN];
```

to provide a default if you should execute the command more than once. Because this definition occurs outside of a function definition, the variable persists even after the command finishes. Variables defined within a function definition (local variables) go away when the function finishes. We copy the file name to `region_file` each time you use the command, and pass it to `get_file()` to provide a default value.

Chapter 7

Epsilon Extension Language

This chapter describes the syntax and semantics of EEL, the Epsilon Extension Language. Starting on page 197, we describe the built-in functions and variables (called *primitives*) of EEL. The tutorial that explains how to compile and load commands into Epsilon begins on page 149. You will find EEL very similar to the C programming language. A list of differences between EEL and C appears on page 187.

7.1   EEL Command Line Flags

To invoke the EEL compiler, type eel *filename*. If you omit the file name, the compiler will display a message showing its command line options.

Before the *filename*, you can optionally specify one or more command line switches. The EEL compiler looks for an environment variable named EEL before examining its command line, then "types in" the contents of that variable before the compiler's real command line. Under Windows, the EEL compiler uses a registry entry named EEL (a "configuration variable", as described on page 8), not an environment variable.

The EEL compiler has the following flags:

- **−b** In each file it compiles, EEL by default includes a variable definition that provides the full path to the original source file. It has a name such as `_loaded_eel_file_xyz` for a file `xyz.e`. This flag omits that variable; it's used for compiling all EEL files that are part of Epsilon's standard distribution. Epsilon's list-customizations command uses the information provided by such variables.

- **−d***mac***!***def* This flag defines the textual macro *mac*, giving it the definition *def*, as if you had defined it using the `#define` command. The syntax −d*mac* defines the macro *mac*, giving it the definition (1). You can also use the syntax −d*mac*=*def*, but beware: if you run EEL via a .BAT or .CMD file, the system will replace any ='s with spaces, and EEL will not correctly interpret the flag.

- **−e** This flag tells the compiler to exclude definitions from `#included` files when it writes the bytecode file. This results in smaller bytecode files. You can safely use this flag when compiling EEL files other than epsilon.e that only include the file eel.h, but it's most useful with autoloaded files. Epsilon will signal an error if you call a function using a variable whose definition has been omitted by −e in all loaded bytecode files.

- **−f** This flag makes the compiler act as a filter, reading EEL code from stdin instead of a file, and writing its binary output to stdout. A file name on the command line is still required, but it is used only for error messages and debugging information.

- **−F** This flag makes the compiler write its binary output to stdout instead of a bytecode file.

- **−i***directory* This flag sets the directories to search for files included with the preprocessor #include command. Precede each search directory with −i. If you use several −i flags on the command line, Epsilon will search the directories in the order they appear.
  If you don't specify any search directories, EEL looks for an EPSPATH configuration variable, which should contain a list of directories, and searches in an "include" subdirectory of each directory on the EPSPATH. For example, if EPSPATH is c:\old;d:\new, EEL searches in c:\old\include, then in d:\new\include. Under Windows, the EEL compiler uses a registry entry named EPSPATH (a "configuration variable", as described on page 8), not an environment variable. (In Epsilon for Unix, a missing EPSPATH variable causes EEL to look in /usr/local/epsilon*VER* (where *VER* is replaced by text representing the current version, such as 101 for 10.1), then /usr/local/epsilon and then /opt/epsilon. In other versions, a missing EPSPATH makes EEL skip this step.)
  EEL also searches for included files based on the location of its executable. If the EEL executable is in c:\somedir\bin, EEL uses the default include path c:\somedir\include. EEL's −w flag makes it skip this step. EEL also skips this step under Unix.

EEL always searches the current directory first if the file name in the #include directive appears between quotes. Then, if there are any −i flags, EEL searches in the specified directories. Next, EEL searches based on the executable's location. Finally, if there were no −i flags, EEL searches based on the EPSPATH setting.

−i− Makes the EEL compiler ignore all prior −i flags. This is useful if you use a configuration variable to always provide certain −i flags.

−n Makes the EEL compiler skip displaying its copyright message.

−o*file* Sets the output file. Normally EEL constructs the file name for the bytecode file based on the input file, with the .e extension replaced by ".b", and puts the bytecode file in the current directory.

−p Makes the compiler display a preprocessed version of the file.

−q Suppress warning messages about unused local variables and function parameters.

−s Leave out debugging information from the bytecode file. Such a file takes up less space, and runs a bit faster. If you use this switch, though, you cannot use the debugger on this file, and the debug key Ctrl-⟨Scroll Lock⟩ (except under Windows and Unix) will not work while such a function executes. We compiled the standard system with the −s flag. You may wish to recompile some files without this flag so you can trace through functions and see how they work.

−v Prints a hash mark each time the compiler encounters a function or global variable definition. Use it to follow the progress of the compiler.

−w This flag tells EEL not to search for included files based on the location of the EEL executable. See the description of the −i flag above.

An example using these switches is:

```
eel −s −p −v −dCODE=3 −oout −i/headers source >preproc
```

7.2 The EEL Preprocessor

EEL includes a preprocessor that can do macro substitution on the source text, among other things. You give preprocessor commands by including lines that start with "**#**" in your source text. A backslash character "****" at the end of a line makes the preprocessor command continue to the next line. This section lists the available preprocessor commands.

    `#define` *identifier replacement-text*

This command defines a textual macro named *identifier*. When this identifier appears again in normal text (not in quotes), it is immediately replaced with the characters in the replacement text.

The rules for legal macro names are the same as the rules for identifiers in the rest of EEL: a letter or the underscore character "_", followed by any number of letters, digits, or underscore characters. Identifiers which differ by case are different identifiers, so mabel, maBel, and MABEL could be three different macros. For clarity, it's best to use all upper case names for macros, and avoid such names otherwise.

When the EEL compiler starts, the macro _EEL_ is predefined, with replacement text (1). The macros UNICODE and BUNICODE are also defined, with the same values. You can test for UNICODE to write EEL code that must also compile in older versions of Epsilon without Unicode support.

Note that these textual EEL macros are not related to keyboard macros. Only the EEL compiler knows about textual macros; Epsilon has no knowledge of them. You cannot bind a textual macro to a key, for

example. Keyboard macros can be bound to a key, and the EEL compiler doesn't know anything about them, only the main Epsilon program. To further confuse matters, other editors refer to their extension languages as macro languages, and call all editor extensions "macros". In this manual, we never use the word "macro" to mean an editor extension written in EEL.

> `#define` *identifier(arg1,arg2,arg3,...) replacement-text*

A macro with arguments is like a normal macro, but instances of the identifier in normal text must be followed by the same number of text sections (separated by commas) as there are arguments. Commas inside quotes or parentheses don't separate text sections. Each of these text sections replace the corresponding identifier within the replacement text. For example, the preprocessor changes

```
#define COLOR(fg, bg)   ((fg) + ((bg) << 4))
int modecol=COLOR(8, 3);
int mcol=COLOR(new_col(6,2),name_to_col("green"));
```

to

```
int modecol=((8) + ((3) << 4))
int mcol=((new_col(6,2))+((name_to_col("green"))<<4))
```

The command

> `#undef` *identifier*

removes the effect of a prior `#define` for the rest of a compilation.

The command

> `#include <`*filename*`>`

inserts the text in another file at this point in the source text. `#include`'s may be nested. In the above format, the EEL compiler searches for the file in each of the `#include` directories specified on the command line, or in a default location if none were specified. See page 155.

If you use quote marks (`"  "`) instead of angle brackets (`<  >`) around the file name of the `#include` command, the EEL compiler will first look in the current directory for the file, before searching the `#include` directories as above. With either delimiter, the compiler will ignore attempts to include a single file more than once in a compilation.

The command

> `#tryinclude <`*filename*`>`

is the same as the `#include` command, except it's not an error if EEL cannot locate the specified file. EEL just ignores the command in that case.

The EEL compiler keeps track of the current source file name and line number to provide error messages during compilation, and passes this information along in the bytecode file (unless you used the `-s` command line option to suppress this). Epsilon then uses this information for the EEL debugger and profiler, and displays it when certain errors occur. You can change the compiler's notion of the current line and source file with the command

> `#line` *number* `"`*filename*`"`

This makes the compiler believe the current file is *filename*, and the `#line` command appears on line *number* of it. If the file name is omitted, only the line number is changed.

```
#if constant-expression
    . . . text . . .
#endif
```

The `#if` command permits sections of the source text to be conditionally included. A constant expression (defined on page 181) follows the `#if`. If the value of the constant expression is nonzero, text from this point to a matching `#endif` command is included. Otherwise, that region is ignored. As part of the constant expression, the `defined()` keyword may be used; `defined(XYZ)` evaluates to 1 if a macro named XYZ has been defined, otherwise 0.

```
#if constant-expression
    . . . text . . .
#else
    . . . text . . .
#endif
```

If an `#else` command appears between the `#if` and the `#endif`, the text following the `#else` is ignored whenever the text preceding it is not. In other words, the text following the `#else` is ignored if the constant is nonzero.

```
#if constant-expression
    . . . text . . .
#elif constant-expression
    . . . text . . .
#else
    . . . text . . .
#endif
```

There may also be one or more `#elif` commands before an `#else` or `#endif` command. EEL evaluates each constant expression in turn, and includes the text following the first of these constant expressions that yields a nonzero value, skipping over all remaining commands in that block.

```
#ifdef identifier
    . . . text . . .
#endif
```

```
#ifndef identifier
    . . . text . . .
#endif
```

You can use the `#ifdef` command in place of the `#if` command. It ignores text between the command and a matching `#endif` if the identifier is not currently defined as a textual macro with the `#define` command. The text is included if the macro is defined. The `#ifndef` command is the same, but with the condition reversed. It includes the text only if the macro is undefined. Both commands may have `#else` or `#elif` sections, as with `#if`.

7.3   Lexical Rules

Comments in EEL begin with the characters `/*`, outside of any quotes. They end with the characters `*/`. The sequence `/*` has no effect while inside a comment, nor do preprocessor control lines.

You can also begin a comment with the characters `//`, outside of quotes. This kind of comment continues until the end of the line.

7.3.1   Identifiers

Identifiers in EEL consist of a letter or the underscore character "`_`", followed by any number of letters, digits, or underscore characters. Upper case and lower case characters are distinct to the compiler, so `Ab` and `ab` are different identifiers. When you load an identifier into Epsilon, Epsilon converts underscores "`_`" to hyphens "`-`" and converts identifiers to lower case. For example, when invoking a command that has been defined in an EEL source file as `this_command()`, you type **this-command**. All characters are significant, and no identifier (or any token, for that matter) may be longer than 1999 characters.

The following identifiers are keywords, and you cannot use them for any other purpose:

```
if              switch          struct          static
else            case            union           unsigned
for             default         keytable        enum
do              goto            typedef         color_class
while           sizeof          buffer          save_spot
return          char            window          save_var
break           short           command         spot
continue        int             on              on_exit
user            volatile        zeroed          color_scheme
byte
```

The keywords `on_exit`, `enum`, `unsigned`, and `static` have no function in the current version of EEL, but we reserve them for future use.

7.3.2   Numeric Constants

The term *numeric constant* collectively refers to decimal constants, octal constants, binary constants and hex constants.

A sequence of digits is a *decimal constant*, unless it begins with the digit 0. If it begins with a 0, it is an *octal constant* (base 8). The characters 0x followed by a hexadecimal number are also recognized (the digits 0–9 and the letters a–f or the letters A–F form hexadecimal numbers). These are the *hex constants*. The characters 0b followed by a binary number form a *binary constant*. A binary number contains only the digits 0 and 1.

All numeric constants in EEL are of type int.

7.3.3   Character Constants

Text enclosed in single quotes as in `'a'` is a *character constant*. The type of a character constant is int. Its value is the ASCII code for the character. Instead of a single character, an escape sequence can appear between the quotes. Each escape sequence begins with a backslash, followed by a character from the following table. A backslash followed by any other character represents that character.

The special escape sequences are:

\n	newline character, ^J
\b	backspace character, ^H
\t	tab character, ^I
\r	return character, ^M
\f	form feed character, ^L
yyy	character with ASCII code *yyy* octal
\x*hh*	character with ASCII code *hh* hexadecimal
\u*hhhh*	character with code *hhhh* hexadecimal
\u{*h*}	character with code *h* hexadecimal
\u[*name*]	character with given Unicode *name*

For example, '\'' represents the ' character, '\\' represents the \ character, '\0' represents the null character, and '\n', '\12', and '\x0A', all represent the newline character (whose ASCII code is 12 in octal notation, base 8, and 0A in hexadecimal, base 16).

The \u sequence is followed by four hex digits, while the \x sequence is followed by only two, and so can only represent low-numbered characters. Enclose the hex digits in curly braces to directly mark the end of the hex number; one to four hex digits can appear within. The square bracket syntax recognizes any standard Unicode character name, such as Greek Capital Letter Alpha. These sequences are all equivalent: \u[Greek Capital Letter Alpha], \u0391, \u{391}.

Anywhere a numeric constant is permitted, so is a character constant, and vice versa.

7.3.4   String Constants

Text enclosed in double quote characters (such as "example") is a *string constant*. It produces a block of storage whose type is *array of char*, and whose value is the sequence of characters between the double quotes, with a null character (ASCII code 0) automatically added at the end. All the escape sequences for character constants work here too.

The compiler merges a series of adjacent string constants into a single string constant (before automatically adding a null character at the end). For example, "sample" "text" produces the same single block of storage as "sampletext".

If an EEL file that begins with a UTF-8 signature ("byte order mark"), then Epsilon decodes UTF-8 sequences in character and string constants. This method lets you include Unicode characters directly instead of using escape sequences. If a file does not begin with a UTF-8 signature, Epsilon interprets bytes literally.

7.4   Scope of Variables

Variables may have two different kinds of "lifetimes", or *scopes*. If you declare a variable outside of any function declaration, it is a *global variable*. If you declare it inside a function declaration, it is a *local variable*.

A local variable only exists while the function it is local to (the one you declared it in) is executing. It vanishes when the function returns, and reappears (with some different value) when the function executes later. If you call the function recursively, each call of the function has its own value for the local variable. You may also declare a variable to be local to a block, in which case it exists only while code inside the block is executing. A local variable so declared only has meaning inside the function or block it is local to.

A global variable exists independently of any function. Any function may use it. If functions declared in different source files use the same global variable, the variable must be declared in both source files (or in

files #included by both files) before its first use. If the two files have different initializations for the variable, only the first initialization has effect.

If a local variable has the same name as a global variable, the local masks the global variable. All references in the block to a variable of that name, from the local variable's definition until the end of the block it is defined in, are to the local variable. After the end of the block, the name again refers to the global variable.

You can declare any global variable to be *buffer-specific* using the `buffer` keyword. A buffer-specific variable has a value for each buffer and a default value. The default value is the value the variable has when you create a new buffer (and hence a new occurrence of the buffer-specific variable). When you refer to a buffer-specific variable, you normally refer to the part that changes from buffer to buffer. To refer to the default portion, append ".default" to the variable name. For example, suppose the variable `foo` is buffer-specific. References to `foo` would then refer to the value associated with the current buffer. To refer to the default value, you would use the expression `foo.default`. (The syntax of appending ".default" is available only when writing EEL programs, not when specifying a variable name to set-variable, for example.) When you save Epsilon's state using the write-state command, Epsilon saves only the default value of each buffer variable, not the value for the current buffer.

Global variables may also be declared *window-specific* using the `window` keyword. A window-specific variable has a separate value for each window and a default value. When Epsilon starts from a state file, it uses the default value saved in the state file to set up the first window. When you split a window, the new window's variables start off with the same values as the original window. Epsilon also uses the default value to initialize each new pop-up window. You can append ".default" to refer to the default value of a window-specific variable.

7.5 Data Types

EEL supports a rich set of data types. First there are the *basic types*:

> `int` These are 32 bit signed quantities. These correspond to integers. The value of an int ranges from -2,147,483,648 to 2,147,483,647.
>
> `short` These are like ints, except they are only 16 bits. Thus the value ranges from -32768 to 32767.
>
> `char` These are 16 bit unsigned quantities. They correspond to characters. For example, the buffer primitive `curchar()` returns an object of type char. The values range from 0 to 65535.
>
> `byte` These are 8 bit unsigned quantities.
>
> `spot` These are references to buffer positions. A spot can remember a buffer position in such a way that after inserting or deleting characters in the buffer, the spot will still be between the same two characters. Like pointers, spots can also hold the special value zero. See page 200.

Besides basic types, there is an infinite set of types derived from these. They are defined recursively as follows:

> `pointer` If *t* is some type, then *pointer to t* is also a type. Conceptually, this is the address of some object of type t. When you dereference an object of type *pointer to t*, the result is of type *t*.
>
> `array` If *t* is some type, then *array of t* is also a type.

`structure` If $t_1, \ldots, t_n$ are types, then *structure of* $t_1, \ldots, t_n$ is also a type. Conceptually, a structure is a sequence of objects, where the *j*th object is of type t_j.

`union` If $t_1, \ldots, t_n$ are types, then *union of* $t_1, \ldots, t_n$ is also a type. Conceptually, a union is an object that can be of any of type $t_1, \ldots, t_n$ at different times.

`function` If *t* is a type, then *function returning t* is also a type.

Any function has a type, which is the type of the value it returns. If the function returns no value, it is of int type, but it is illegal to attempt to use the function's value.

Regardless of its type, you may declare any function to be a command (using the `command` keyword) if it takes no parameters. Commands like `named-command` on Alt-X will then complete on its name, but there is no other difference between commands and *subroutines* (user-defined functions which are not commands). Functions that the user is expected to invoke directly (by pressing a key, for example) are generally commands, while functions that act as helpers to commands are generally subroutines. Nothing prevents an EEL function from calling a command directly, though, and the user can invoke any subroutine directly as well (providing that it takes no arguments). Though a command may not have arguments, it may return a value (which is ignored when the user directly invokes it).

7.5.1 Declarations

Declarations in EEL associate a type with an identifier. The structure of EEL declarations mimics the recursive nature of EEL types.

A *declaration* is of the form:

declaration:
    type-specifier ;
    type-specifier declarator-list ;
declarator-list:
    declarator
    declarator , *declarator-list*

A *type specifier* names one of the basic types, a structure or union (described on page 165), or a typedef, a type abbreviation (described on page 167).

type-specifier:
    `char`
    `short`
    `int`
    `struct` *struct-or-union-specifier*
    `union` *struct-or-union-specifier*
    `spot`
    typedef-name
typedef-name:
    identifier

A *declarator*, on the other hand, specifies the relationship of the identifier being declared to the type named by the type specifier. If this is a recursive type, the relationship of the identifier's type to the basic type of the type specifier is indicated by the form of the declarator.

Declarators are of the following form:

declarator:
    identifier
    (*declarator*)
    ∗ *declarator*
    declarator [*constant-expression*]
    declarator [ ]
    declarator ()

If *D* is a declarator, then (*D*) is identical to *D*. Use parentheses to alter the binding of composed declarators. We discuss this more on page 166.

7.5.2   Simple Declarators

In the simplest case, the identifier being declared is of one of the basic types. For that, the declarator is simply the identifier being declared. For example, the declarations

```
int length;
char this_character;
short small_value;
```

declare the type of the identifier `length` to be int, the type of `this_character` to be char, and the type of `small_value` to be short.

If the relationship between the identifier and the type specified in the type specifier is more complex, so is the declarator. Each type of declarator in the following sections contains exactly one identifier, and that is the identifier being declared.

7.5.3   Pointer Declarators

Pointer declarators are used in conjunction with type specifiers to declare variables of type *pointer to t*, where *t* is some type. The form of a pointer declarator is

    ∗ *declarator*

Suppose *T* is a type specifier and *D* is a declarator, and the declaration "*T D*;" declares the identifier embedded in *D* to be of type ". . . *T*". Then the declaration *T *D;* declares the identifier in *D* to be of type ". . . *pointer to T*". Several examples illustrate the concept.

```
int l;
int *lptr;
int **ldblptr;
```

Clearly, the first declaration declares `l` to be of type int. The type specifier is `int` and the declarator is `l`.

The second line is a little more complicated. The type specifier is still `int`, but the declarator is `*lptr`. Using the rule above, we see that `lptr` is a pointer to an int. This is immediately clear from the above if you substitute "int" for *T*, and "lptr" for *D*.

Similarly, the third line declares `ldblptr` to be a pointer to a pointer to an int.

7.5.4   Array Declarators

Array declarators are used in conjunction with type specifiers to declare objects of type *array of t*, where *t* is some type. The form of an array declarator is

  declarator [*constant-expression*]

but you may omit the constant expression if

- An *initialized* global variable of type "*array of* . . ." is being defined. (See page 168.) In this case, the first constant-expression may be omitted, and the size of the array will be calculated from the initializer.

- A *function argument* (sometimes called a formal parameter) of type "*array of* . . ." is being declared. Since the type of the argument will be changed to "*pointer to* . . ." (as described on page 186) the first constant-expression may be omitted.

The rules for constant expressions appear on page 181.

Suppose *T* is a type specifier and *D* is a declarator, and the declaration "*T D;*" declares the identifier embedded in *D* to be of type ". . . *T*". Then the declaration *T (D)[ ];* declares the identifier to be of type ". . . *array of T*".

As an example, consider:

```
int (one_dim)[35];
int ((two_dim)[35])[44];
```

The first line declares the identifier `one_dim` to be of type *array of int*.

The second line declares `two_dim` to be *array of array of int*. Clearly, we can have arbitrary multi-dimensional arrays by declaring the arrays in this manner.

As another example, consider the following:

```
char (*arg);
char (*argptr)[5];
char *(argary[5]);
```

From the preceding section, we know that the first line declares `arg` to be a pointer to a char. From this section, we see that the second line declares `argptr` to be of type *pointer to array of char*.

Compare this to the third line, which declares `argary` to be of type *array of pointer to char*.

When you have mixed declarators as you have in this example, you sometimes can elide parentheses according to the precedence rules of declarators. See section 7.5.7 for these precedences.

7.5.5   Function Declarators

Function declarators are used in conjunction with type specifiers to declare variables of type *function returning t*, where *t* is some type. The form of a function declarator is

  declarator ()

or

  declarator (*ansi-argument-list*)

Again, suppose *T* is a type specifier and *D* is a declarator, and the declaration "*T D;*" declares the identifier embedded in *D* to be of type "... *T*". Then the declaration *T* (*D*) (); declares the identifier to be of type "... *function returning T*".

Consider:

```
char (c)();
char *(fpc());
char (*pfc)(int count, char *msg);
```

The first line declares c to be of type *function returning char*. The second line declares fpc to be a *function returning pointer to char*. The third line declares pfc to be of type *pointer to function returning char*. The third example also declares that pfc requires two parameters and gives their types; the first two examples provide no information about their functions' parameters.

7.5.6 Structure and Union Declarations

This section describes how to define variables of type *structure of* $t_1, \ldots, t_n$, where $t_1, \ldots, t_n$ are each types. First, we give an informal description, with examples, of how structures are often declared. A more formal description with BNF diagrams follows.

There is a special type-specifier, called a *structure-or-union specifier*, that defines structure and union types. This type-specifier has several forms.

The simplest form is seen in the following example:

```
struct {
    int field1;
    char name[30];
    char *data;
}
```

The field names of the structure are the identifiers being declared within the curly braces. These declarations look like variable declarations, but instead of declaring variables, they declare *field names*. The type of a particular field is the type the identifier would have if the declaration were a variable declaration.

The example above refers to a structure with fields named field1, name, and data, with types *int*, *array of char*, and *pointer to char*, respectively.

Use the structure-or-union specifier like the other type-specifiers (int, short, char, and spot) in declarations. For example:

```
struct {
    int field1;
    char name[30];
    char *data;
} rec, *recptr, recary[4];
```

declares rec to be a structure variable, recptr to be a pointer to a structure, and recary to be an array of (4) structures.

The structure-or-union-specifier may contain a *tag*, which gives a short name for the entire structure. For example, the type-specifier in the following example:

```
struct recstruct {
    int field1;
    char name[30];
    char *data;
};
```

creates a new type, `struct recstruct`, that refers to the structure being defined. Given this structure tag, we may define our structure variables in the following manner:

```
struct recstruct rec, *recptr, recary[4];
```

Structure (or union) tags also let you create self-referential types. Consider the following:

```
struct list {
    int data;
    struct list *next;
};

struct list list1, list2;
```

This creates a structure type `list`, which has a `data` field that's an int, and a `next` field that is a pointer to a `list` structure. A structure may not contain an instance of itself, but may contain (as in this example) a pointer to an object of its type.

More formally, a structure-or-union-specifier has the following form:

struct-or-union-specifier:
    struct-or-union-tag
    struct-or-union-tag { *member-list* }
    { *member-list* }
struct-or-union-tag:
    identifier
member-list:
    type-specifier declarator-list ;
    type-specifier declarator-list ; *member-list*

A description of how to use structures and unions in expressions appears on page 181.

7.5.7   Complex Declarators

As some of the examples thus far have shown, you can compose (combine) declarators to yield arbitrarily complicated types, like *function returning pointer to an array of 10 chars*:

```
char (*foo())[10];
```

When composing declarators, function and array declarators have the same precedence. They each take precedence over pointer declarators. So the example we used in section 7.5.5:

```
char *(fpc());
```

could have been written more simply as

```
char *fpc();.
```

The rule that EEL follows for declarations is that the identifier involved is to be declared so that an expression with the form of the declarator has the type of the type specifier. This implies that the grouping of operators in a declarator follows the same rules as the operators do in an expression.

There are a few restrictions on the combinations of declarators when functions are involved (and so on the combinations of types). Functions may not return arrays, structures, unions, or functions, but they may return pointers to any of these. Similarly, functions may not be members of structures, unions, or arrays, but pointers to functions may be.

7.5.8   Typedefs

typedef-definition:
> typedef *type-specifier declarator-list* ;

You can use typedefs to provide convenient names for complicated types. Once you define it, use a typedef as a type specifier (like int) in any declaration. A typedef definition looks just like a variable definition, except that the keyword typedef appears before the type specifier. The name of the typedef being defined appears instead of the variable name, and the typedef has the same type the variable would have had.

Typedefs only serve as abbreviations. They always create types that could be made in some other way. A variable declared using a typedef is just the same as a variable declared using the full specification. For example:

```
typedef short *NAME_LIST;
NAME_LIST nl, narray[20];
```

is equivalent to

```
short *nl, *narray[20];
```

7.5.9   Type Names

EEL's sizeof operator and its casting operator specify particular types using *type names*. A type name looks like a declaration of a single variable, except that the variable name is missing (as is the semicolon at the end). For example, int * is a type name referring to a pointer to an int.

type-name:
> *type-specifier abstract-declarator*

abstract-declarator:
> *empty*
> (*abstract-declarator*)
> * *abstract-declarator*
> *abstract-declarator* [*constant-expression*]
> *abstract-declarator* [ ]
> *abstract-declarator* ( )
> *abstract-declarator* (*ansi-argument-list*)

Note that you could interpret a type name like int *() in two ways: either as a function returning a pointer to an int (like int *foo();) or as a pointer to an int (like int *(foo);). EEL rules out the latter

by requiring that a parenthesized *abstract-declarator* be nonempty. Given this, the system is not ambiguous, and an identifier can appear in only one place in each type name to make a legal declaration.

The same precedence rules apply to type names as to normal declarators (or to expressions). For example, the type name `char *[10]` refers to an array of 10 pointers to characters, but `char (*)[10]` refers to a pointer to an array of 10 characters.

7.6   Initialization

Declarations for the formal parameters of functions work just as described above, but you can additionally provide local and global variables with a specific initial value.

> *local-variable-definition:*
> > *type-specifier local-declarator-list* ;
>
> *local-declarator-list:*
> > *local-declarator*
> > *local-declarator* , *local-declarator-list*
>
> *local-declarator:*
> > *declarator*
> > *declarator* = *expression*

You can initialize a local variable with any expression so long as the corresponding assignment would be permitted. Since you cannot assign to variables with types such as "*array of . . .*" and "*structure of . . .*", you cannot initialize such local variables at compile time. Local variables (those defined within a block) have undefined initial values if no explicit initialization is present.

> *global-variable-definition:*
> > *type-specifier global-declarator-list* ;
> > *global-modifier-list global-declarator-list* ;
> > *global-modifier-list type-specifier global-declarator-list* ;
>
> *global-modifier-list:*
> > *global-modifier*
> > *global-modifier global-modifier-list*
>
> *global-modifier:*
> > `buffer`
> > `window`
> > `zeroed`
> > `user`
> > `volatile`
>
> *global-declarator-list:*
> > *global-declarator*
> > *global-declarator* , *global-declarator-list*
>
> *global-declarator:*
> > *declarator*
> > *declarator* = *string-constant*
> > *declarator* = *initializer*
>
> *initializer:*
> > *constant-expression*

> { *initializer-list* }
> { *initializer-list* , }
> *initializer-list:*
> > *initializer*
> > *initializer* , *initializer-list*

You may initialize a global variable of type "array of characters" with a string constant. If you omit the length of the array in a declaration with such an initialization, it's set to just contain the initializing string (including its terminating null character).

If no explicit initialization is specified, variables defined globally are set to zero. If you provide a partial initialization (for example, if you specify the first 5 characters in a 10 character array), the remainder of the variable is set to zero. Initializers for global variables must involve only constant expressions known at compile time, whereas initializers for local variables may involve arbitrary expressions (including function calls, for example).

When Epsilon loads a file defining an initialized global variable and the variable was already defined to have the same type, the initialization has no effect: the variable's value remains the same. If the new declaration specifies a different type for the variable, however, the variable's value is indeed changed. (Actually, Epsilon only compares the sizes of the variables. If you redefine an integer as a four character array, Epsilon won't apply the new initialization.) For example, suppose you declare `foo` to be an int and initialize it to 5. If you later load a file which redeclares `foo` to be an int and initializes it to 7, the value of `foo` would remain 5. If instead you redeclare `foo` to be a char and reinitialize it to `'C'`, then the value would change, since the size of a char is different from the size of an int.

To tell Epsilon that it must reinitialize the variable each time it reads a definition, use the `volatile` keyword. Every time you load a bytecode file containing such a variable definition, Epsilon will set the variable according to its initialization.

If you declare a global variable that is a number, spot, or pointer, the initializer must be a constant expression. In fact, if the variable is a spot or pointer, you can only initialize it with the constant zero. For example:

```
int i=3;
char *name="harold";
```

initializes the int variable `i` to be 3, and the character pointer `name` to point to the first character in the string "harold". The variable `name` must be a local variable. If it were global, then you could initialize it only to zero, which is equivalent to not initializing it at all (see above).

If you declare a global array, you can initialize each element of the array. The initializer in this case would be a sequence of constant expressions, separated by commas, with the whole thing enclosed in braces {}. Consider the following examples:

```
int ary1[4] = { 10, 20, 30, 40 };
int ary2[ ] = { 10, 20, 30, 40 };
int ary3[4] = { 10, 20 };
```

Here we have `ary1` declared to be an array of 4 ints. We initialize the first element in the array to 10, the second to 20, and so on. The declaration of `ary2` does the same thing. Notice that the square brackets in the declarator are empty. The EEL compiler can tell from the initializer that the size must be 4. The declaration of `ary3` specifies the size of the array, but only initializes the first two elements. The compiler initializes the remaining two elements to zero.

The initializers for global structures are similar. The items between the curly braces are a sequence of expressions, with each expression's type matching the type of the corresponding field name. For example, the declaration:

```
struct {
    int f1;
    char f2;
    short f3;
} var = { 33, 't', 22 };
```

declares the variable `var` to be a structure with fields `f1`, `f2`, and `f3`, with types *int*, *char*, and *short* respectively. The declaration initializes the `f1` to 33, the character field `f2` to `'t'`, and the short field `f3` to 22.

You cannot initialize either unions or local structures. Global pointers may only be initialized to zero (which is equivalent to not initializing them at all).

If you initialize an array or structure which has subarrays or substructures, simply recursively apply the rules for initialization. For example, consider the following:

```
struct {
    char c;
    int ary1[3];
} var = { 't',  { 3, 4, 5} };
```

This declares `var` to be a structure containing a character and an array of 3 ints. It initializes the character to `'t'`, and the array of ints so that the first element is 3, the second 4, and the third 5.

7.7   Statements

EEL has all of the statements of the C programming language. You can precede a statement by a *label*, an identifier followed by a colon, which you can use with the `goto` statement to explicitly alter the flow of control. Except where noted below, statements are executed in order.

7.7.1   Expression Statement

expression;

The expression is simply evaluated. This is the form of function calls and assignments, and is the most common type of statement in EEL.

7.7.2   If Statement

if (*expression*)
    statement

If the value of *expression* is not zero, *statement* executes. Otherwise control passes to the statement after the `if` statement.

if (*expression*)
    statement1

```
else
      statement2
```

If the value of *expression* is not zero, *statement1* executes. If the value of *expression* is zero, control passes to *statement2*.

7.7.3   While, Do While, and For Statements

```
while ( expression )
      statement
```

In a `while` loop, the *expression* is evaluated. If nonzero, the *statement* executes, and the expression is evaluated again. This happens over and over until the expression's value is zero. If the expression is zero the first time it is evaluated, *statement* is not executed at all.

```
do
      statement
while ( expression);
```

A `do while` loop is just like a plain `while` loop, except the statement executes *before* the expression is evaluated. Thus, the statement will always be evaluated at least once.

```
for ( expression1; expression2; expression3 )
      statement
```

In a `for` loop, first *expression1* is evaluated. Then *expression2* is evaluated, and if it is zero EEL leaves the loop and begins executing instructions after *statement*. Otherwise the statement is executed, *expression3* is evaluated, and *expression2* is evaluated again, continuing until *expression2* is zero.

You can omit any of the expressions. If you omit *expression2*, it is like *expression2* is nonzero. `while` (*expression*) is the same as `for` (; *expression*;). The syntax `for` (;;) creates an endless loop that must be exited using the `break` statement (or one of the other statements described below).

7.7.4   Switch, Case, and Default Statements

```
switch ( expression )
      statement
```

`case` *constant-expression*: *statement*

`default`: *statement*

Statements within the *statement* following the `switch` (which is usually a block, as described below) are labeled with constant expressions using `case`. The *expression* is evaluated (it must yield an int), and Epsilon branches to the `case` statement with the matching constant. If there is no match, Epsilon branches to the `default` statement if there is one, and skips over the `switch` statement if not.

A `case` or `default` statement associates with the smallest surrounding `switch` statement. Each `switch` statement must have at most one `case` statement with any given value, and at most one `default` statement.

7.7.5   Break and Continue Statements

```
break;
```

This statement exits from the smallest containing `for`, `while`, `do while` or `switch` statement. The `break` statement must be the last statement in each `case` if you don't want execution to "fall through" and execute the statements for the following cases too.

```
continue;
```

The `continue` statement immediately performs the test for the smallest enclosing `for`, `while`, or `do while` statement. It is the same as jumping to the end of the *statement* in each of their definitions. In the case of `for`, *expression3* will be evaluated first.

7.7.6   Return Statement

```
return;
```

```
return expression;
```

The `return` statement exits from the function it appears in. The first form returns no value, and produces an error message if you called the function in a way that requires a value. The second form returns *expression* as the value of the function. It must have the same type as you declared the function to be. It is not an error for the value to be unused by the caller.

If execution reaches the end of a function definition, it is the same as if `return;` were there.

7.7.7   Save_var and Save_spot Statements

statement:
    `save_var` *save-list*;
    `save_spot` *save-list*;
save-list:
    save-item
    save-item , *save-list*
save-item:
    identifier
    identifier = *expression*
    identifier modify-operator expression
    identifier ++
    identifier −−

The `save_var` statement tells Epsilon to remember the current value of a variable, and set it back to its current value when the function that did the `save_var` exits. Epsilon will restore the value no matter how the function exits, even if it calls another function which signals an error, and this aborts out of the calling function.

You can provide a new value for the variable at the same time you save the old one. Epsilon first saves the old value, then assigns the new one. You can use any of the assignment operators listed on page 179, as well as the `++` and `--` operators.

For example, this command plays a note at 440 Hz for one second, without permanently changing the user's variable settings for the bell (in versions of Epsilon that support changing the bell's frequency and duration).

```
command play_note()
{
    save_var beep_frequency = 440;
    save_var beep_duration = 100;
    ding();      /* uses beep_ variables */
}
```

The `save_spot` statement functions like `save_var`, but it creates a *spot* (see page 200) in the current buffer to hold the old value. The spot will automatically go away when the function exits. Use `save_spot` instead of `save_var` when you wish to save a buffer position, and you want it to stay in the right place even if the buffer contents change.

The `save_var` and `save_spot` statements can apply to global variables with "simple" types: those that you can directly assign to with the = operator. They don't work on structures, for example, or on local variables.

Although the `save_var` and `save_spot` statements resemble variable declarations, they are true statements. You can use the `if` statement (above), for example, to only save a variable in certain cases. These statements operate with a "stack" of saved values, so that if you save the same variable twice in a function, only the first setting will have an effect on the final value of the variable. (Repeated save statements take up space on the saved value stack, however, so they should be avoided.) When you save a buffer-specific or window-specific variable, Epsilon remembers which buffer's or window's value was saved, and restores only that one.

The `restore_vars()` primitive restores all variables saved in the current function. After a `restore_vars()`, future modifications to any saved variables won't be undone.

7.7.8 On_exit Statement

statement:

> `on_exit` *statement*

An `on_exit` statement tells Epsilon to run some code later, when the current function exits. It can be used to clean up temporary data. As with the `save_var` statement in the previous section, Epsilon will run the specified code no matter how the function exits, even if it calls another function which signals an error, and this aborts out of the calling function.

The statement to be executed later can use local or global variables, call other functions, and so forth. It can be a block or other complex type of statement, but it cannot use certain control flow statements: `switch`, `case`, `default`, `break`, `continue`, `return`, `goto`, or labels. And it cannot use `on_exit`, `save_var`, `save_spot`, or `restore_vars()` itself. (If the `on_exit` statement's code calls a function, that function can use any of these.)

The `restore_vars()` primitive also causes all pending `on_exit` statements to be executed at once, just as if the current function were about to exit.

All `on_exit`, `save_var`, and `save_spot` statements push their pending operations onto a stack, and they are executed in order when the function returns or when `restore_vars()` is used, newest to oldest.

7.7.9 Goto and Empty Statements

> `goto` *label*;

> `label:` *statement*

The next statement executed after the `goto` will be the one following the *label*. It must appear in the same function as the `goto`, but may be before or after.

```
;
```

This null statement is occasionally used in looping statements, where all the "work" of the loop is done by the expressions. For example, a loop that calls a function `foo()` repeatedly until it returns zero can be written as

```
while (foo()) ;.
```

7.7.10 Block

```
{
```
declarations
statements
```
}
```

Anywhere you can have a statement, you can have a *block*. A block contains any number of local variable *declarations* or *statements* (including zero). The variables declared in the block are local to the block, and you may only use them in the following *statements* (or in statements contained in those statements). A block's declarations can be mixed in freely among its statements. The body of a function definition is itself a block.

7.8 Conversions

When a value of a certain type is changed to another type, a *conversion* occurs.

When a number of some type is converted to another type of number, if the number can be represented in the latter type its value will be unchanged. All possible characters can be represented as ints or short ints, and all short ints can be represented as ints, so these conversions yield unchanged values.

Technically, Epsilon will sign-extend a short int to convert it to an int, but will pad a character with zero bits on the left to convert it to an int or short int. Converting a number of some type to a number of a shorter type is always done by dropping bits.

A pointer may not be converted to an int, or vice versa, except for function pointers. The latter may be converted to a short int, or to any type that a short int may be converted to. A pointer to one type may be converted to a pointer to another type, as long as neither of them is a function pointer.

All operators that take numbers as operands will take any size numbers (characters, short ints, or ints). The operands will be converted to int if they aren't already ints. Operators that yield numbers always produce ints.

7.9 Operator Grouping

In an expression like

```
10  op1  20  op2  30
```

the compiler determines the rules for grouping by the *precedence* and *associativity* of the operators *op1* and *op2*. Each operator in EEL has a certain precedence, with some precedences higher than others. If *op1* and

Highest Precedence

l-to-r	()	[ ]	->	.
r-to-l	All unary operators (see below)			
l-to-r	*	/	%	
l-to-r	+	–		
l-to-r	<<	>>		
l-to-r	>	<	>=	<=
l-to-r	==	!=		
l-to-r	&			
l-to-r	^			
l-to-r	|			
l-to-r	&&			
l-to-r	||			
l-to-r	?	:		
r-to-l	All assignment operators (see below)			
l-to-r	,			

Lowest Precedence

Assignment operators are: =   *=   /=   %=   +=   –=
                           <<=   >>=   &=   ^= |=

Unary operators are: *   &   –   !   ~
                    ++   --   sizeof   (*type-name*)

Figure 7.1: Operator Precedence

op2 have different precedences, the one with the higher precedence groups tighter. In table 7.1, operators with higher precedences appear on a line above operators with lower precedences. Operators with the same precedence appear on the same line.

For example, say *op1* is + and *op2* is *. Since *'s line appears above +'s, * has a higher precedence than + and the expression 10 + 20 * 30 is the same as 10 +(20 * 30).

If two operators have the same precedence, the compiler determines the grouping by their associativity, which is either left-to-right or right-to-left. All operators of the same precedence have the same associativity. For example, suppose *op1* is – and *op2* is +. These operators have the same precedence, and associate left-to-right. Thus 10 – 20 + 30 is the same (10 – 20) + 30. All operators on the same line in the table have the same precedence, and their associativity is given with either "l-to-r" or "r-to-l."

Enclosing an expression in parentheses alters the grouping of operators. It does not change the value or type of an expression itself.

7.10   Order of Evaluation

Most operators do not guarantee a particular order of evaluation for their operands. If an operator does, we mention that fact in its description below. In the absence of such a guarantee, the compiler may rearrange

calculations within a single expression as it wishes, if the result would be unchanged ignoring any possible side effects.

For example, if an expression assigns a value to a variable and uses the variable in the same expression, the result is undefined unless an operator that guarantees order of evaluation occurs at an appropriate point.

Note that parentheses do not alter the order of evaluation, but only serve to change the grouping of operators. Thus in the statement

```
i = foo() + (bar() + baz());
```

the three functions may be called in any order.

7.11 Expressions

7.11.1 Constants and Identifiers

expression:
  numeric-constant
  string-constant
  identifier
  `color_class` *identifier*

The most basic kinds of expressions are numeric and string constants. Numeric constants are of type "int", and string constants are of type "array of character". However, EEL changes any expression of type "array of . . ." into a pointer to the beginning of the array (of type "pointer to . . ."). Thus a string constant results in a pointer to its first character.

An identifier is a valid expression only if it has been previously declared as a variable or function. A variable of type "array of . . ." is changed to a pointer to the beginning of the array, as described above.

Some expressions are called *lvalue expressions*. Roughly, lvalue expressions are expressions that refer to a changeable location in memory. For example, if `foo` is an integer variable and `func()` is a function returning an integer, then `foo` is an lvalue, but `func()` is not. The `&` and `.` operators, the `++` and `--` operators, and all assignment operators require their operands to be lvalues. Only the `*`, `[ ]`, `->`, and `.` operands return lvalues.

An identifier which refers to a variable is an lvalue if its type is an integer, a spot, a pointer, a structure, or a union, but not if its type is an array or function.

If an identifier has not been previously declared, and appears in a function call as the name of the function, it is implicitly declared to be a function returning an int.

If the name of a previously declared function appears in an expression in any context other than as the function of a function call, its value is a function pointer to the named function. Function pointers may not point to primitive functions.

For example, if `foo` is previously undeclared, the statement `foo(1, 2);` declares it as a function returning an int. If the next statement is `return foo;`, a pointer to the function `foo()` will be returned.

Once a color class `newclass` has been declared, you can refer to it by using the special syntax `color_class newclass`. This provides a numeric code that refers to the particular color class. It's used in conjunction with the primitives `alter_color()`, `add_region()`, `set_character_color()`, and others. See page 90 for basic information on color classes, and page 183 for information on declaring color classes in EEL.

7.11.2 Unary Operators

expression:
>   ! *expression*
>   ∗ *expression*
>   & *expression*
>   – *expression*
>   ˜ *expression*
>   sizeof *expression*
>   sizeof(*type-name*)
>   (*type-name*) *expression*
>   ++ *expression*
>   –– *expression*
>   *expression* ++
>   *expression* ––

The ! operator yields one if its operand is zero, and zero otherwise. It can be applied to pointers, spots, or numbers, but its result is always an int.

The unary ∗ operator takes a pointer and yields the object it points to. If its operand has type "pointer to . . .", the result has type ". . .", and is an lvalue. You can also apply ∗ to an operand of type "spot", and the result is a number (a buffer position).

The unary & operator takes an lvalue and returns a pointer to it. It is the inverse of the ∗ operator, and its result has type "pointer to . . ." if its operand has type ". . .". (You cannot construct a spot by applying the & operator to a position. Use the alloc_spot() primitive described on page 200.)

The unary – and ˜ operators work only on numbers. The first negates the given number, and the second flips all its bits, changing ones to zeros and zeros to ones.

The sizeof operator yields the size in bytes of an object. You can specify the object as an expression or with a type name (described on page 167). In the latter case, sizeof returns the size in bytes of an object of that type. Characters and shorts require two bytes, and ints four bytes. An array of 10 ints requires 40 bytes, and this is the number sizeof(int [10]) will give, not 10.

An expression with a parenthesized type name before it is a *cast*. The cast converts the expression to the named type using the rules beginning on page 174, and the result is of that type. Specify the type using a type name, described on page 167.

The ++ and –– operators increment and decrement their lvalue operands. If the operator appears before its operand, the value of the expression is the new value of the operand. The expression (++var) is the same as (var += 1), and (--var) is the same as (var -= 1). You can apply these operators to pointers, in which case they work as described under pointer addition below.

If the ++ or –– operators appear after their operand, the operand is changed in the same way, but the value of the expression is the value of the operand *before* the change. Thus the expression var++ has the same value as var, but var has a different value when you reference it the next time.

7.11.3 Simple Binary Operators

expression:
>   *expression* + *expression*
>   *expression* – *expression*

       expression ∗ *expression*
       expression / *expression*
       expression % *expression*
       expression == *expression*
       expression != *expression*
       expression < *expression*
       expression > *expression*
       expression <= *expression*
       expression >= *expression*
       expression && *expression*
       expression || *expression*
       expression & *expression*
       expression | *expression*
       expression ^ *expression*
       expression << *expression*
       expression >> *expression*

The binary + operator, when applied to numbers, yields the sum of the numbers. One of its operands may also be a pointer to an object in an array. In this case, the result is a pointer to the same array, offset by the number to another object in the array. For example, if p points to an object in an array, p + 1 points to the next object in the array and p - 1 points to the previous object, regardless of the object's type.

The binary - operator, when applied to numbers, yields the difference of the numbers. If the first operand is a pointer and the second is a number, the rules for addition of pointers and numbers apply. For example, if p is a pointer, p - 3 is the same as p + -3.

Both operands may also be pointers to objects in the same array. In this case the result is the difference between them, measured in objects. For example, if arr is an array of ten ints, p1 points to the third int, and p2 points to the eighth, then p1 - p2 yields the int -5. The result is undefined if the operands are pointers to different arrays.

The binary ∗ operator is for multiplication, and the / operator is for division. The latter truncates toward 0 if its operands are positive, but the direction of truncation is undefined if either operand is negative. The % operator provides the remainder of the division of its operands, and x % y is always equal to x - (x / y) ∗ y. All three operators take only numbers and yield ints.

The == operator yields one if its arguments are equal and zero otherwise. The arguments must either both be numbers, both spots, or both pointers to objects of the same type. However, if one argument is the constant zero, the other may be a spot or any type of pointer, and the expression yields one if the pointer is null, and zero otherwise. The != operator is just like the == operator, but returns one where == would return zero, and zero where == would return one. The result of either operator is always an int.

The <, >, <=, and >= operators have a value of one when the first operand is less than, greater than, less than or equal to, or greater than or equal to (respectively) the second operand. The operands may both be numbers, they may be pointers to the same array, or one may be a pointer or spot and the other zero. In the last case, Epsilon returns values based on the convention that a null pointer or spot is equal to zero and a non-null one is greater than zero. The result is undefined if the operands are pointers to different arrays of the same type, and it is an error if they are pointers to different types of objects, or if one is a spot and the other is neither a spot nor zero.

The && operator yields one if both operands are nonzero, and zero otherwise. Each operand may be a pointer, spot, or number. Moreover, the first operand is evaluated first, and if it is zero, the second operand will not be evaluated. The result is an int.

The || operator yields one if either of its operands are nonzero, and zero if both are zero. Each operand may be a pointer, spot, or number. The first operand is evaluated first, and if it is nonzero, the second operand will not be evaluated. The result is an int.

The & operator yields the bitwise AND of its numeric operands. The | and ^ operators yields the bitwise OR and XOR (respectively) of their numeric operands. The result for all three is an int. A bit in the result of an AND is on if both corresponding bits in its operands are on. A bit in the result of an OR is on if either of the corresponding bits in its operands are on. A bit in the result of an XOR is on if one of the corresponding bits in its operands is on and the other is off.

The << operator yields the first operand with its bits shifted to the left the number of times given by the right operand. The >> operator works similarly, but shifts to the right. The former fills with zero bits, and the latter fills with one bits if the first operand was negative, and zero bits otherwise. If the second operand is negative or greater than 31, the result is undefined. Both operands must be numbers, and the result is an int.

7.11.4 Assignment Operators

expression:
> *expression = expression*
> *expression modify-operator expression*

modify-operator:
> `+=`
> `−=`
> `*=`
> `/=`
> `%=`
> `&=`
> `|=`
> `^=`
> `<<=`
> `>>=`

The plain assignment operator = takes an lvalue (see page 176) as its first operand. The object referred to by the lvalue is given the value of the second operand. The types of the operands may both be numbers, spots, pointers to the same type of object, or compatible structures. If the first operand is a pointer or spot and the second is the constant zero, the pointer or spot is made null. The value of the expression is the new value of the first operand, and it has the same type.

The other kinds of assignment operators are often used simply as abbreviations. For example, if a is a variable, a += (b) is the same as a = a + (b). However, the first operand of an assignment is only evaluated once, so if it has side effects, they will only occur once.

For example, suppose a is an array of integers with values 10, 20, 30, and so forth. Suppose p() is a function that will return a pointer to the first element of a the first time it's called, then a pointer to the second element, and so forth. After the statement *p() += 3;, a will contain 13, 20, 30. After *p() = *p() + 3;, however, a is certain not to contain 13, 20, 30, since p() will never return a pointer to the same element of a twice. Because the order of evaluation is unspecified with these operators, the exact result of the latter statement is undefined (either 10, 13, 30 or 23, 20, 30).

The result of all these assignment statements is the new value of the first operand, and will have the same type. The special rules for mixing pointers and ints with the + and − operators also apply here.

7.11.5 Function Calls

> *expression:*
>> *expression* `()`
>>
>> *expression* `(` *expression-list* `)*
>
> *expression-list:*
>> *expression*
>>
>> *expression* `,` *expression-list*

An expression followed by a parenthesized list of expressions (arguments) is a function call. Usually the first expression is the name of a function, but it can also be an expression yielding a function. (The only operator that yields a function is the unary ∗ when applied to a function pointer.) The type of the result is the type of the returned value. If the function returns no value, the expression must appear in a place where its value is not used. You may call any function recursively.

If an identifier that has not been previously declared appears as the name of the function, it is implicitly declared to be a function returning an int.

Each argument is evaluated and a copy of its value is passed to the function. Character and short arguments are converted to ints in the process. Aside from this, the number and type of arguments must match the definition of the function. The order of evaluation of the arguments to a function is undefined.

Since only a copy of each parameter is passed to the function, a simple variable cannot be altered if its name only appears as the argument to a function. To alter a variable, pass a pointer to it, and have the function modify the object pointed to. Since an array is converted to a pointer whenever its name occurs, an array that is passed to a function can indeed be altered by the function. Numbers, spots, and pointers may be parameters, but structures, unions, or functions cannot be. Pointers to such things are allowed, of course.

An EEL function can call not just other EEL functions, but also any of Epsilon's built-in functions, known as primitives. These are listed in the next chapter. An EEL function can also call a keyboard macro as a function. The word "function" refers to any of the various types of routines that a command written in EEL can call. These include other commands or subroutines (themselves written in EEL), primitives that are built into Epsilon and cannot be changed, and keyboard macros (see page 124). Textual macros that are defined with the `#define` preprocessor statement are *not* functions.

Each function may require a certain number of arguments and may return a value of a particular type. Keyboard macros, however, never take arguments or return a value.

7.11.6 Miscellaneous Operators

> *expression:*
>> *expression* `?` *expression* `:` *expression*
>>
>> *expression* `,` *expression*
>>
>> *expression* `[` *expression* `]`
>>
>> *expression* `->` *identifier*
>>
>> *expression* `.` *identifier*

The conditional operator `? :` has three operands. The first operand is always evaluated first. If nonzero, the second operand is evaluated, and that is the value of the result. Otherwise, the third operand is evaluated, and that is the value of the result. Exactly one of the second and third operands is evaluated. The first operand may be a number, spot, or pointer. The second and third operands may either both be numbers, both spots, both pointers to the same type of object, or one may be a pointer or spot and the other the constant

zero. In the first case the result is an int, and in the last two cases the result is a spot or a pointer of the same type.

The `,` operator first evaluates its first argument and throws away the result. It then evaluates its second argument, and the result has that value and type. In any context where a comma has a special meaning (such as in a list of arguments), EEL assumes that any commas it finds are used for that special meaning.

The `[ ]` operator is EEL's subscripting operator. Because of the special way that addition of a pointer and a number works, we can define the subscripting operator in terms of other operators. The expression `e1[e2]` is the same as `*((e1)+(e2))`, and since addition is commutative, also the same as `e2[e1]`. In practice, subscripting works in the expected way. Note that the first object in an array has subscript 0, however. One of the operands must be a pointer and the other a number. The type of the result is that of the pointed-to object.

The `.` operator disassembles structures or unions. Its operand is an lvalue which is a structure or union. After the `.` an identifier naming one of the operand's members must appear. The result is an lvalue referring to that member.

The `->` operator is an abbreviation for a dereference (unary `*`) followed by a member selection as above. Its operand is a pointer to a structure or union, and it is followed by the name of one of the structure's or union's members. The result is an lvalue referring to that member. The expression `strptr->membername` is the same as the expression `(*strptr).membername`.

7.12   Constant Expressions

A constant expression is an expression which does not contain certain things. It may not have references to variables, string constants, or function calls. No subexpressions may have a type of spot, structure, union, array, or pointer. It may have numeric constants, character constants, and any operators that act on them, and the `sizeof` operator may appear with any operand.

Additionally, for constant expressions in preprocessor lines, you can test if a macro m has been defined by writing `defined(m)`. This expression evaluates to `1` if a macro by that name has been defined, `0` if not.

The term "the constant zero" means a constant expression whose value is zero, not necessarily a numeric constant.

7.13   Global Definitions

program:
   global-definition
   global-definition program
global-definition:
   function-definition
   global-variable-definition
   keytable-definition
   typedef-definition
   color-class-definition

Each file of EEL code consists of a series of definitions for global variables and functions. Global variable definitions have the same format as local variable definitions. The first definition of a global variable Epsilon receives determines the initial value of the variable, and later initializations have no effect, unless you use the `volatile` keyword when defining the variable (see page 169). If the first definition provides no explicit initialization, the variable is filled with zeros or null pointers as appropriate, depending on its type.

You can declare any global variable (except a key table or color class) to be buffer-specific by placing the keyword `buffer` before the type specifier. When the definition is first read in, its initializer determines the value of the variable for each buffer that then exists, and also the default value of the variable. Whenever you create a new buffer (and hence a new copy of the buffer-specific variable), the variable's value in that buffer is set to the default value.

Similarly, you can declare any global variable except a key table or color class to be window-specific by placing the keyword `window` before the type specifier. When the definition is first read in, its initializer determines the value of the variable for each window that then exists, and also the default value of the variable. Whenever you split a window in two, the new window inherits its initial value for the window-specific variable from the original window. Epsilon uses the default value of a window-specific variable when it creates the first tiled window while starting up, and when it creates pop-up windows.

Epsilon's write-state command writes a new state file containing all variables, EEL functions, macros, colors, and so forth that Epsilon knows about. The file includes the current values of all numeric variables, all global character array variables, and any structures or unions containing just these types. But Epsilon doesn't save the values of variables containing pointers or spots, and sets these to zero as it writes a state file. You can put the `zeroed` keyword before the definition of a variable of any type to tell Epsilon to zero that variable when it writes a state file.

In commands like set-variable, Epsilon distinguishes between user variables and system variables, and only shows the former in its list of variables you can set. By default, each global variable you define is a system variable that users will not see. Put the `user` keyword before a variable's definition to make the variable a user variable.

7.13.1   Key Tables

keytable-definition:
    `keytable` *keytable-list* ;
keytable-list:
    identifier
    identifier , *keytable-list*

A key table is a set of bindings, one for each key on the keyboard, with keys modified by control, alt, and shift counted as separate keys. Various mouse actions and system events are also represented by special key codes. Each entry in the key table contains a short integer, which is an index into the name table. In other words, each entry corresponds to a named Epsilon object, either a command, subroutine, keyboard macro, or another key table.

You can declare a key table by using the `keytable` keyword in place of the type specifier in a global variable definition. A key table definition can contain no initialization, just `keytable` followed by a list of comma-separated key table names and a semicolon. A key table acts like an array of short ints, but you can also use it in the `on` part of a function definition (as described below).

Key codes, the values that index a key table, can be very large numbers. Looping through all possible key codes with a simple `for (i = 0; i < MAXKEYS; i++)` statement is far too slow; see page 326 for the right way to write such loops.

7.13.2   Color Classes

color-class-definition:
    `color_class` *color-class-list* ;
    `color_scheme` *color-scheme-list* ;

color-class-list:
    color-class-item
    color-class-item , *color-class-list*
color-class-item:
    identifier
    identifier `color_scheme` *string-constant color-pair*
    identifier { *color-scheme-spec-list* }
    identifier color-pair
color-scheme-spec-list:
    color-scheme-spec
    color-scheme-spec color-scheme-spec-list
color-scheme-spec:
    `color_scheme` *string-constant color-pair* ;
color-scheme-list:
    color-scheme-item
    color-scheme-item , *color-scheme-list*
color-scheme-item:
    string-constant
    string-constant `color_class` *identifier color-pair*
    string-constant { *color-class-spec-list* }
color-class-spec-list:
    color-class-spec
    color-class-spec color-class-spec-list
color-class-spec:
    `color_class` *identifier color-pair* ;
color-pair:
    = `color_class` *identifier*
    constant-expression
    constant-expression `on` *constant-expression*

A color class specifies a particular pair of foreground and background colors Epsilon should use on a certain part of the screen, or when displaying a certain type of text. For example, Epsilon uses the color class `c_keyword` to display keywords in C-like languages. More precisely, the color class specifies which foreground/background pair of colors to display under each defined color scheme. If the user selects a different color scheme, Epsilon will immediately begin displaying C keywords using the `c_keyword` color pair defined for the new scheme.

Before you use a color class in an expression like `set_character_color(pos1, pos2, color_class c_keyword);`, you must declare the color class (outside of any function definition) using the `color_class` keyword:

```
color_class c_keyword;
```

When you declare a new color class, you may wish to specify the colors to use for a particular color scheme using the `color_scheme` keyword:

```
color_class c_keyword
    color_scheme "standard-gui" black on white;
color_class c_keyword
    color_scheme "standard-color" green on black;
```

If you have many color definitions all for the same color class, you can use this syntax:

```
color_class c_keyword {
    color_scheme "standard-gui" black on white;
    color_scheme "standard-color" green on black;
};
```

Similarly, if you have many color definitions for the same color scheme, you can avoid repeating it by writing:

```
color_scheme "standard-gui" {
    color_class c_keyword black on white;
    color_class c_function blue on white;
    color_class c_identifier black on white;
};
```

To specify the particular foreground and background colors for a color class (using the syntax *foreground* on *background*), you can use these macros defined in eel.h:

```
#define black           MAKE_RGB(0, 0, 0)
#define dark_red        MAKE_RGB(128, 0, 0)
#define dark_green      MAKE_RGB(0, 128, 0)
#define brown           MAKE_RGB(128, 128, 0)
// etc.
```

See that file for the current list of named colors. These functions use the MAKE_RGB() macro, providing particular values for red, green, and blue. You can use this macro yourself, in a color class definition, to specify precise colors:

```
color_scheme "my-color-scheme" {
    color_class c_keyword MAKE_RGB(223, 47, 192) on yellow;
};
```

There are several other macros useful in color definitions:

```
#define MAKE_RGB(rd,grn,bl)  ((rd) + ((grn) << 8) + ((bl) << 16))
#define GETRED(rgb)          ((rgb) & 0xff)
#define GETGREEN(rgb)        (((rgb) >> 8) & 0xff)
#define GETBLUE(rgb)         (((rgb) >> 16) & 0xff)
```

The GETRED(), GETGREEN(), and GETBLUE() macros take a color expression created with MAKE_RGB() and extract one of its three components, which are always numbers from 0 to 255.

The foreground color for a color class may also include font style bits, by or'ing any of the macros EFONT_BOLD, EFONT_UNDERLINED, and EFONT_ITALIC into the color code.

The ETRANSPARENT macro is a special code that may be used in place of a background color. It tells Epsilon to substitute the background color of the "text" color class in the current color scheme. The following three examples are all equivalent:

```
color_class text color_scheme "standard-gui" yellow on red;
color_class c_keyword color_scheme "standard-gui" blue on red;

color_class text color_scheme "standard-gui" yellow on red;
color_class c_keyword color_scheme "standard-gui" blue
    on ETRANSPARENT;

color_class text color_scheme "standard-gui" yellow on red;
color_class c_keyword color_scheme "standard-gui" blue;
```

The last example works because you may omit the on *background* part from the syntax *foreground* on *background*, and just specify a foreground color. Epsilon interprets this as if you typed on `transparent`, and substitutes the background color specified for `"text"`.

You can also specify that a particular color class is the same as a previously-defined color class, like this:

```
color_scheme "standard-gui" {
    color_class text black on white;
    color_class tex_text = color_class text;
};
```

When, for the current scheme, there's no specific color information for a color class, Epsilon looks for a default color class specification, one that's not associated with any scheme:

```
color_class diff_added black on yellow;
color_class c_string cyan;
color_class c_charconst = color_class c_string;
```

The first definition above says that, in the absence of any color-scheme-specific setting for the `diff_added` color class, it should be displayed as black text on a yellow background. The second says that text in the `c_string` color class should be displayed using cyan text, on the default background for the scheme (that defined for the `text` color class). And the third says that text in the `c_charconst` color class should be displayed the same as text in the `c_string` color class for that scheme.

Internally, Epsilon stores all color class settings that occur outside any color scheme in a special color scheme, which is named `"color-defaults"`. See page 247 for more on colors.

7.13.3 Function Definitions

function-definition:
    function-head block
    function-head argument-decl-list block
    ansi-function-head block
    callable-function-head block
callable-function-head:
    typed-function-head
    `command` *typed-function-head*
    typed-function-head `on` *binding-list*
    `command` *typed-function-head* `on` *binding-list*
binding-list:

       keytable-name [*constant-expression*]
       keytable-name [*constant-expression*] **,** *binding-list*
keytable-name:
       identifier
typed-function-head:
       identifier **()**
       type-specifier identifier **()**
function-head:
       identifier **(** *argument-list* **)**
       type-specifier identifier **(** *argument-list* **)**
ansi-function-head:
       identifier **(** *ansi-argument-list* **)**
       type-specifier identifier **(** *ansi-argument-list* **)**
ansi-argument-list:
       type-specifier declarator
       type-specifier declarator **,** *ansi-argument-list*
argument-list:
       identifier
       identifier **,** *argument-list*
argument-decl-list:
       type-specifier declarator-list **;**
       type-specifier declarator-list **;** *argument-decl-list*

    A function definition begins with a type specifier, the name of the function, and parentheses surrounding a comma-separated list of arguments. Any bindings may be given here using the on keyword, as described below. Declarations for the arguments then appear, and the body of the function follows. If the command keyword appears before the type specifier, the function is a command, and Epsilon will do completion on the function when it asks for the name of a command. A function may be a command only if it has no arguments.

    You may omit the type specifier before the function name, in which case the function's type is int. You may also omit the declaration for any argument, in which case the argument will be an int. Note that unlike some languages such as Pascal, if there are no arguments, an empty pair of parentheses must still appear, both in the definition and where you call the function.

    You may also define functions using ANSI C/C++ syntax, in which type information for function arguments appears with the argument names inside parentheses. These function headers have the same effect:

```
average(int count, short *values)      average(count, values)
                                       short *values;
```

    When you call a function, arguments of type char or short are automatically changed to ints. A corresponding change happens to declarations of function arguments and return values. Additionally, function arguments declared as an array of some type are changed to be a pointer to the same type, just as array variables are changed to pointers to the start of the array when their names appear in expressions (see page 176). For example, these two function headers have the same effect.

```
short average(count, values)           average(count, values)
char count;                            short *values;
short values[ ];
```

The user can call any function which takes no arguments, or bind such a function to a key. Functions which are normally invoked in this way can be made commands with the `command` keyword, but this is not necessary. If you omit the `command` keyword, Epsilon will not perform command completion on the function's name. The `on` keyword can appear after the (empty) parentheses of a function's argument list, to provide bindings for the function. Each binding consists of a key table name, followed by a constant (the key number) in square brackets `[ ]`. There may be several bindings following the `on` keyword, separated by commas. You must have previously declared the key table name in the same file (or an `#included` file). The binding takes effect when you load the function.

Sometimes it is necessary to declare an identifier as a function, although the function is actually defined in a separately compiled source file. For example, you must declare a function before you use a pointer to that function. Also, the EEL compiler must know that a function returns a non-numeric type if its return value is used. Any declaration of an identifier with type *function returning . . .* is a function declaration. Function declarations may appear anywhere a local or global variable declaration is legal. So long as the identifier is not masked by a local variable of the same name, the declaration has effect until the end of the file.

Any function named `when_loading()` is automatically executed when you load the bytecode file it appears in into Epsilon. There may be any number of `when_loading()` functions defined in a file, and they execute in order, while the file is being loaded. Such functions are deleted as soon as they return. They may take no arguments.

7.14 Differences Between EEL And C

- Global variables may not be initialized with any expression involving pointers. This includes strings, which may only be used to directly initialize a declared array of characters. That is,

```
char example[ ] = "A string.";
```

is legal, while

```
char *example = "A string.";
```

is not.

- There are no static variables or functions. All local variables vanish when the function returns, and all global objects have names that separately compiled files can refer to.

- The C reserved word "extern" does not exist. In EEL, you may define variables multiple times with no problems, as long as they are declared to have the same type. The first definition read into Epsilon provides the initialization of the variable, and further initializations have no effect. However, if the variable is later declared with a different size, the size changes and the new initialization takes effect. To declare a function without defining it in a particular source file, see page 187.

- The C types "long", "enum", "void", "float", and "double" do not exist. Ints and shorts are always signed. Chars and bytes are always unsigned. There are no C bit fields. The C reserved words "long", "float", and "double" are not reserved in EEL.

- EEL provides the basic data type `spot`, and understands color class expressions and declarations using the `color_class` and `color_scheme` keywords.

- You may not cast between pointers and ints, except that function pointers may be cast to shorts, and vice versa. The constant zero may be cast to any pointer type. A pointer may be cast to a pointer of another type, with the exception of function pointers.

- You can use the reserved word `keytable` to declare empty key tables, as in

  ```
  keytable reg_tab, cx_tab;
  ```

  Local key tables are not permitted.

- The reserved word `command` is syntactically like a storage class. Use it to indicate that the function is normally called by the user, so command completion will work. The user can also call other functions (as long as they have no arguments) but the completion facility on command names ignores them.

- After the head of any function definition with no arguments, you can use the reserved word `on` to give a binding. It is followed by the name of a key table already declared, and an index (constant int expression) in square brackets. There may be more than one (separated by commas). For example,

  ```
  command visit_file() on cx_tab[CTRL('V')]
  ```

- You can use the reserved word `buffer` as a storage class for global variables. It declares a variable to have a different value for each buffer, plus a default value. As you switch between buffers, a reference to a buffer-specific variable will refer to a different value.

- You can also use the reserved word `window` as a storage class for global variables. This declares the variable to have a different value for each window, plus a default value. As you switch between windows, a reference to a window-specific variable will refer to a different value.

- The reserved words `zeroed` and `user` do not exist in C. See page 182. The reserved word `volatile` does exist in ANSI C, but serves a different purpose in EEL. See page 169.

- The EEL statements `save_var`, `save_spot`, and `on_exit` do not exist in C. See page 172.

- In each compile, an include file with a certain name is only read once, even if there are several `#include` directives that request it.

7.15   Syntax Summary

program:
    global-definition
    global-definition program
global-definition:
    function-definition
    global-variable-definition
    keytable-definition
    typedef-definition
    color-class-definition
typedef-definition:
    `typedef` *type-specifier declarator-list* ;
color-class-definition:
    `color_class` *color-class-list* ;
    `color_scheme` *color-scheme-list* ;
color-class-list:
    color-class-item
    color-class-item , *color-class-list*

color-class-item:
    identifier
    identifier `color_scheme` *string-constant color-pair*
    identifier { *color-scheme-spec-list* }
    identifier color-pair
color-scheme-spec-list:
    color-scheme-spec
    color-scheme-spec color-scheme-spec-list
color-scheme-spec:
    `color_scheme` *string-constant color-pair* ;
color-scheme-list:
    color-scheme-item
    color-scheme-item , color-scheme-list
color-scheme-item:
    string-constant
    string-constant `color_class` *identifier color-pair*
    string-constant { *color-class-spec-list* }
color-class-spec-list:
    color-class-spec
    color-class-spec color-class-spec-list
color-class-spec:
    `color_class` *identifier color-pair* ;
color-pair:
    constant-expression
    constant-expression `on` *constant-expression*
keytable-definition:
    `keytable` *keytable-list* ;
keytable-list:
    identifier
    identifier , *keytable-list*
global-variable-definition:
    type-specifier global-declarator-list ;
    global-modifier-list global-declarator-list ;
    global-modifier-list type-specifier global-declarator-list ;
global-modifier-list:
    global-modifier
    global-modifier global-modifier-list
global-modifier:
    `buffer`
    `window`
    `zeroed`
    `user`
    `volatile`
declarator-list:
    declarator

       declarator , *declarator-list*
declarator:
    identifier
    (*declarator*)
    * *declarator*
    declarator [*constant-expression*]
    declarator [ ]
    declarator ()
global-declarator-list:
    global-declarator
    global-declarator , *global-declarator-list*
global-declarator:
    declarator
    declarator = *string-constant*
    declarator = *initializer*
initializer:
    constant-expression
    string-constant
    { *initializer-list* }
    { *initializer-list* , }
initializer-list:
    initializer
    initializer , *initializer-list*
type-specifier:
    `char`
    `short`
    `int`
    `struct` *struct-or-union-specifier*
    `union` *struct-or-union-specifier*
    `spot`
    typedef-name
typedef-name:
    identifier
struct-or-union-specifier:
    struct-or-union-tag
    struct-or-union-tag { *member-list* }
    { *member-list* }
struct-or-union-tag:
    identifier
member-list:
    type-specifier declarator-list ;
    type-specifier declarator-list ; *member-list*
type-name:
    type-specifier abstract-declarator
abstract-declarator:

empty

(*abstract-declarator*)

∗ *abstract-declarator*

abstract-declarator [*constant-expression*]

abstract-declarator [ ]

abstract-declarator ( )

abstract-declarator (*ansi-argument-list*)

function-definition:

function-head block

function-head argument-decl-list block

ansi-function-head block

callable-function-head block

callable-function-head:

typed-function-head

command *typed-function-head*

typed-function-head on *binding-list*

command *typed-function-head* on *binding-list*

binding-list:

keytable-name [*constant-expression*]

keytable-name [*constant-expression*] , *binding-list*

keytable-name:

identifier

typed-function-head:

identifier ( )

type-specifier identifier ( )

function-head:

identifier (*argument-list*)

type-specifier identifier (*argument-list*)

ansi-function-head:

identifier (*ansi-argument-list*)

type-specifier identifier (*ansi-argument-list*)

ansi-argument-list:

type-specifier declarator

type-specifier declarator , *ansi-argument-list*

argument-list:

identifier

identifier , *argument-list*

argument-decl-list:

type-specifier declarator-list ;

type-specifier declarator-list ; *argument-decl-list*

block:

{ *statement-list* }

{ }

local-variable-definition:

type-specifier local-declarator-list ;

local-declarator-list:
 local-declarator
 local-declarator , *local-declarator-list*
local-declarator:
 declarator
 declarator = *expression*
statement-list:
 statement
 statement statement-list
statement:
 expression ;
 `if` (*expression*) *statement*
 `if` (*expression*) *statement* `else` *statement*
 `while` (*expression*) *statement*
 `do` *statement* `while` (*expression*);
 `for` (*opt-expression* ; *opt-expression* ; *opt-expression*) *statement*
 `switch` (*expression*) *statement*
 `case` *constant-expression* : *statement*
 `default`: *statement*
 `break;`
 `continue;`
 `return;`
 `return` *expression* ;
 `save_var` *save-list* ;
 `save_spot` *save-list* ;
 `on_exit` *statement*
 `goto` *label* ;
 label : *statement*
 ;
 local-variable-definition
 typedef-definition
 block
save-list:
 save-item
 save-item , *save-list*
save-item:
 identifier
 identifier = *expression*
 identifier modify-operator expression
 identifier ++
 identifier --
label:
 identifier
opt-expression:
 empty

    expression

expression:

    numeric-constant

    string-constant

    identifier

    identifier `.default`

    `color_class` *identifier*

    `(` *expression* `)`

    `!` *expression*

    `*` *expression*

    `&` *expression*

    `-` *expression*

    `~` *expression*

    `sizeof` *expression*

    `sizeof(` *type-name* `)`

    `(` *type-name* `)` *expression*

    `++` *expression*

    `--` *expression*

    expression `++`

    expression `--`

    expression `+` *expression*

    expression `-` *expression*

    expression `*` *expression*

    expression `/` *expression*

    expression `%` *expression*

    expression `==` *expression*

    expression `!=` *expression*

    expression `<` *expression*

    expression `>` *expression*

    expression `<=` *expression*

    expression `>=` *expression*

    expression `&&` *expression*

    expression `||` *expression*

    expression `&` *expression*

    expression `|` *expression*

    expression `^` *expression*

    expression `<<` *expression*

    expression `>>` *expression*

    expression `=` *expression*

    expression modify-operator expression

    expression `?` *expression* `:` *expression*

    expression `,` *expression*

    expression `()`

    expression `(` *expression-list* `)`

    expression `[` *expression* `]`

   expression . *identifier*
   expression -> *identifier*
modify-operator:
   `+=`
   `-=`
   `*=`
   `/=`
   `%=`
   `&=`
   `|=`
   `^=`
   `<<=`
   `>>=`
expression-list:
   expression
   expression , *expression-list*
constant-expression:
   numeric-constant
   (*constant-expression*)
   ! *constant-expression*
   - *constant-expression*
   ~ *constant-expression*
   `sizeof` *constant-expression*
   `sizeof`(*type-name*)
   constant-expression + *constant-expression*
   constant-expression - *constant-expression*
   constant-expression * *constant-expression*
   constant-expression / *constant-expression*
   constant-expression % *constant-expression*
   constant-expression == *constant-expression*
   constant-expression != *constant-expression*
   constant-expression < *constant-expression*
   constant-expression > *constant-expression*
   constant-expression <= *constant-expression*
   constant-expression >= *constant-expression*
   constant-expression && *constant-expression*
   constant-expression || *constant-expression*
   constant-expression & *constant-expression*
   constant-expression | *constant-expression*
   constant-expression ^ *constant-expression*
   constant-expression << *constant-expression*
   constant-expression >> *constant-expression*
   constant-expression ? *constant-expression* : *constant-expression*
   constant-expression , *constant-expression*

Chapter 8

Primitives and EEL Subroutines

In this chapter, we describe all EEL primitives, as well as a few useful EEL subroutines. In Epsilon, the term "primitive" refers to a function or variable that is not written or defined in EEL, but rather built into Epsilon.

Each section discusses items that pertain to a particular topic, and begins with EEL declarations for the items discussed in that section. If we implemented an item as an EEL subroutine, the declaration often includes a comment that identifies the EEL source file defining the item.

Some EEL primitives have optional parameters. For example, you can call the `get_tail()` primitive as either `get_tail(fname, 1)` or `get_tail(fname)`. Any missing parameter automatically takes a value of zero. In this manual, we indicate an optional parameter by showing a ? before it.

When writing EEL extensions, an easy way to look up the documentation on the primitive or subroutine at point is to press F1 F ⟨Enter⟩.

8.1 Buffer Primitives

8.1.1 Changing Buffer Contents

```
insert(int ch)
user buffer int point;
```

An Epsilon *buffer* contains text that you can edit. Most of the primitives in this section act on, or refer to, one of the buffers designated as the *current buffer*.

The `insert()` primitive inserts a single character into the current buffer. Its argument says what character to insert. The buffer's insertion point, or just point, refers to the particular position in each buffer where insertions occur.

The int variable named `point` stores this position. Its value denotes the number of characters from the beginning of the buffer to the spot at which insertions happen. For example, a value of zero for `point` means that insertions would occur at the beginning of the buffer. A value of one for `point` means that insertions would occur after the first character, etc.

To change the insertion point, you can assign a new value to `point`. For example, the statement

```
point = 3;
```

makes insertions occur after the third character in the buffer, assuming the buffer has at least 3 characters. If you set `point` to a value less than zero, `point` takes the value zero. Similarly, if you set `point` to a value greater than the size of the buffer, its value becomes the number of characters in the buffer.

When the current buffer changes, the value of the variable `point` automatically changes with it. We call variables with this behavior *buffer-specific* variables. See page 296.

```
int size()
```

The primitive function `size()` returns the number of characters in the current buffer. You cannot set the size directly: you can change the size of the buffer only by inserting or deleting characters. For this reason, we implemented `size()` as a function, not a variable like `point`.

The variable `point` refers not to a character position, but rather to a character boundary, a place between characters (or at the beginning or end of a buffer). The legal values for point range from zero to `size()`. We will refer to a value in this range, inclusive of the ends, as a *position*. A position is a place between characters in a buffer, or at the beginning of the buffer, or at the end. The value of a position is the number of characters before it in the buffer. In EEL, ints (integers) hold positions.

When Epsilon inserts a character, it goes before point, not after it. If Epsilon didn't work this way, inserting a, then b, then c would result in cba, not abc.

```
delete(int pos1, int pos2)
int delete_if_highlighted()
```

The `delete()` primitive deletes all characters between the two positions supplied as arguments to it. The order of the arguments doesn't matter.

The `delete()` primitive doesn't save deleted text in a kill buffer. The kill commands themselves manage the kill buffers, and use the `delete()` primitive to actually remove the text.

Commands that insert text often begin by calling the `delete_if_highlighted()` subroutine. If there's a highlighted region, this subroutine deletes it and returns 1. Otherwise (or if the `typing-deletes-highlight` variable has been set to zero), it returns 0.

```
replace(int pos, int ch)
int character(int pos)
int curchar()
```

The `replace()` primitive changes the character at position `pos` to `ch`. The parameter `pos` refers to the position before the character in question. Therefore, the value of `pos` can range from 0 to `size()-1`, inclusively.

The `character()` primitive returns the character after the position specified by its argument, `pos`. The `curchar()` returns the same value as `character(point)`. These two primitives return −1 when the position involved isn't valid, such as at the end of the buffer or before its start (when `pos` is less than zero). For example, `character(size())` returns −1, as does `curchar()` with point at the end of the buffer.

```
stuff(char *str)
int bprintf(char *format, ...)
int buffer_printf(char *name, char *format, ...)
int buf_printf(int bnum, char *format, ...)
buf_stuff(int bnum, char *s, int len)
```

The `stuff()` function inserts an entire string into the current buffer.

The `bprintf()` function also inserts a string, but it takes a format string plus other arguments and builds the string to insert using the rules on page 235. The `buffer_printf()` functions similarly, except that it takes the name of the buffer into which to insert the string. It creates the buffer if necessary. Similarly, `buf_printf()` takes a buffer number, and inserts the formatted string into that buffer. All of the primitives described in this paragraph return the number of characters they inserted into the buffer.

The `buf_stuff()` primitive includes a length parameter, so it can handle text that may contain null characters. It inserts the `len` characters at `s` into the buffer `buf`.

8.1.2 Moving Text Between Buffers

```
xfer(char *buf, int from, int to)
buf_xfer(int bnum, int from, int to) /* buffer.e */
raw_xfer(int bnum, int from, int to)
buf_xfer_colors(int bnum, int from, int to)
grab_buffer(int bnum)        /* buffer.e */
```

The `xfer()` subroutine transfers characters from one buffer to another. It copies the characters between `from` and `to` in the current buffer and inserts them at point in the named buffer. It positions the `mark` in the

named buffer just before the inserted characters, and positions its `point` right after the insertion. The current buffer doesn't change. The `buf_xfer()` subroutine works similarly, but accepts a buffer number instead of a name. Both use the `raw_xfer()` primitive to transfer the text.

The `buf_xfer_colors()` subroutine is like `buf_xfer()`, but copies any colors set by `set_character_color()` as well.

The `grab_buffer()` subroutine copies text in the other direction. It inserts the text of buffer number `bnum` into the current buffer before point, setting the mark before the inserted text.

8.1.3 Getting Text from a Buffer

```
grab(int pos1, int pos2, char *to)
grab_expanding(int pos1, int pos2, char **toptr, int minlen)
buf_grab_bytes(int buf, int from, int to, char *dest)
```

The `grab()` primitive copies characters from the buffer to a string. It takes the range of characters to copy, and a character pointer indicating where to copy them. The buffer doesn't change. The positions may be in either order. The resulting string will be null-terminated.

The `grab_expanding()` subroutine is similar, but works with a dynamically allocated character pointer, not a fixed-length character array. Pass a pointer to a `char *` variable, and the subroutine will resize it as needed to hold the result. The `char *` variable may hold NULL initially. The `minlen` parameter provides a minimum allocation length for the result.

The `buf_grab_bytes()` subroutine copies characters in the specified range in the buffer `buf` into the character array `dest`, in the same fashion as `grab()`. Despite its name, it operates on 16-bit characters, not 8-bit bytes.

```
grab_full_line(int bnum, char *str) /* buffer.e */
grab_line(int bnum, char *str)      /* buffer.e */
```

The `grab_full_line()` subroutine copies the entire current line of buffer number `bnum` into the character array `str`. It doesn't change point. The `grab_line()` subroutine copies the remainder of `bnum`'s current line to `str`, and moves to the start of the next line. Neither function copies the ⟨Newline⟩ at the end of the line, and each returns the number of characters copied.

```
grab_line_offset(int b, char *s, int offset, int wrap)
```

The `grab_line_offset()` subroutine copies a line from buffer `b` into `s`, discarding any trailing newline and returning the string's length. The `offset` parameter specifies which line: 0 means the buffer's current line, 1 means the next, and so forth. The subroutine moves to end of the appropriate line. If `offset` is negative, the subroutine moves to a previous line and copies it, remaining at its start. The `wrap` parameter, if nonzero, makes the subroutine wrap around to the opposite end of the buffer if it hits an end when counting lines; if zero, a too-high `offset` returns zero and empties `s`.

```
int grab_numbers(int bnum, int *nums)    /* buffer.e */
int break_into_numbers(char *s, int *nums)
```

The `grab_numbers()` subroutine uses `grab_line()` to retrieve a line from buffer `bnum`. Then it breaks the line into words (separated by spaces and tabs), and tries to interpret each word as a number by calling the `numtoi()` subroutine. It puts the resulting numbers in the array `nums`. The function returns the number of words on the line.

The `break_into_numbers()` subroutine is similar, but retrieves the numbers from a string `s`. It returns the count of numbers it found, putting their values into the `nums` array.

```
int grab_string(int bnum, char *s, char *endmark) /* buffer.e */
int grab_string_expanding(int bnum, char **s,
                          char *endmark, int minlen)
```

The `grab_string()` subroutine copies from buffer `bnum` into `s`. It copies from the buffer's current position to the beginning of the next occurrence of the text `endmark`, and leaves the buffer's point after that text. It returns `1`, unless it couldn't find the `endmark` text. In that case, it moves to the end of the buffer, sets `s` to the empty string, and returns `0`.

The `grab_string_expanding()` subroutine is similar, but works with dynamically allocated character pointers, not fixed-length character arrays. Pass a pointer to a `char *` variable, and the subroutine will resize it as needed to hold the result. The `char *` variable may hold NULL initially. The `minlen` parameter provides a minimum allocation length for the result.

8.1.4   Spots

```
spot alloc_spot(?int left_ins)
free_spot(spot sp)
int spot_to_buffer(spot sp)
```

A place in the buffer is usually recorded and saved for later use as a count of the characters before that place: this is a position, as described on page 197. Sometimes it is important for the stored location to remain between the same pair of characters even if many changes are made to other parts of the buffer (affecting the number of characters before the saved location).

Epsilon provides a type of variable called a *spot* for this situation. The declaration

```
        spot sp;
```

says that sp can refer to a spot. It doesn't create a new spot itself, though.

The `alloc_spot()` primitive creates a new spot and returns it, and the `free_spot()` primitive takes a spot and discards it. The spot that `alloc_spot()` returns is initially set to point, and is associated with the current buffer. Deleting a buffer frees all spots associated with it. If you try to free a spot whose buffer has already been deleted, Epsilon will ignore the request, and will not signal an error.

The `spot_to_buffer()` primitive takes a spot and returns the buffer number it was created for, or `-1` if the buffer no longer exists, or `-2` if the buffer exists, but that particular spot has since been deleted.

If the `left_ins` parameter to `alloc_spot()` is nonzero, a left-inserting spot is created. If the `left_ins` parameter is `0`, or is omitted, a right-inserting spot is created. The only difference between the two types of spots is what they do when characters are inserted right where the spot is. A left-inserting spot stays after such inserted characters, while a right-inserting spot stays before them. For example, imagine an empty buffer, with all spots at 0. After five characters are inserted, any left-inserting spots will be at the end of the buffer, while right-inserting spots will remain at the beginning.

A spot as returned by `alloc_spot()` behaves a little like a pointer to an int, in that you must dereference it by writing `*sp` to obtain the position it currently refers to. For example:

```
fill_all()      /* fill paragraphs, leave point alone */
{
spot oldpos = alloc_spot(), oldmark = alloc_spot();

*oldpos = point;
*oldmark = mark;        /* save old values */
point = 0;              /* make region be whole buffer */
mark = size();
fill_region();          /* fill paragraphs in region */
mark = *oldmark;        /* restore values */
point = *oldpos;
free_spot(oldmark);     /* free saving places */
free_spot(oldpos);
}
```

A simpler way to write the above subroutine uses EEL's `save_spot` keyword. The `save_spot` keyword takes care of allocating spots, saving the original values, and restoring those values when the subroutine exits. See page 172 for more on `save_spot`.

```
fill_all()      /* fill paragraphs, leave point alone */
{               /* uses save_spot */
save_spot point = 0;    /* make region be whole buffer */
save_spot mark = size();
fill_region();          /* fill paragraphs in region */
}
```

Like a pointer, a spot variable can contain zero, and `alloc_spot()` is guaranteed never to return this value. Epsilon signals an error if you try to dereference a spot which has been freed, or whose buffer no longer exists.

```
buffer spot point_spot;
buffer spot mark_spot;
#define point    *point_spot
#define mark     *mark_spot
/* These variables are actually defined
   differently.  See below. */
```

Each new buffer begins with two spots, `point_spot` and `mark_spot`, set to the beginning of the buffer. `Point_spot` is a left-inserting spot, while `mark_spot` is a right-inserting spot. These spots are created automatically with each new buffer, and you cannot free them. You can think of the built-in variables `point` and `mark` as simply macros that yield `*point_spot` and `*mark_spot`, respectively. That's why you don't need to put a `*` before each reference to `point`.

```
user buffer int point;       /* True definitions */
user buffer int mark;
spot get_spot(int which)
#define point_spot      get_spot(0)
#define mark_spot       get_spot(1)
```

Actually, while `point` and `mark` could be defined as macros, as above, they're not. Epsilon recognizes them as built-in primitives for speed. On the other hand, `point_spot` and `mark_spot` actually are macros! They use the `get_spot()` primitive, which has no function other than to return these two values.

```
do_set_mark(int val)
```

The `do_set_mark()` subroutine sets the current buffer's mark to the specified value. It also records the current virtual column (which, typically, should match the mark). The rectangle commands retrieve this, so that in virtual mode you can copy rectangles that end in virtual space.

```
set_spot(spot *s, int pos)
```

The `set_spot()` subroutine sets a spot so it refers to the position `pos` in the current buffer. Pass it the address of a spot variable. If the spot is zero or refers to a different buffer, the subroutine will create a new right-inserting spot in the current buffer, freeing the old spot.

8.1.5 Narrowing

```
user buffer int narrow_start;
user buffer int narrow_end;
int narrow_position(int p)        /* buffer.e */
```

Epsilon provides two primitive variables, `narrow_start` and `narrow_end`, that restrict access to the current buffer. The commands narrow-to-region and widen-buffer, described on page 143, use these variables. Epsilon ignores the first `narrow_start` characters and the last `narrow_end` characters of the buffer. Usually, these variables have a value of zero, so no such restriction takes place. Characters outside of the narrowed region will not appear on the screen, and will remain outside the control of normal Epsilon commands.

If you try to set a primitive variable such as `point` to a position outside of the narrowed area, Epsilon will change the value to one inside the narrowed area. For example, suppose the buffer contains one hundred characters, with the first and last ten characters excluded, so only eighty appear on the screen. In this case, `size()` will return one hundred, and `narrow_start` and `narrow_end` will each have a value of ten. The statement `point = 3;` will give `point` a value of ten (the closest legal value), while the statement `point = 10000;` will give `point` the value ninety. Epsilon adjusts the parameters of primitive functions in the same way. Suppose, in the example above, you try to delete all the characters in the buffer, using the `delete()` primitive. Epsilon would take the statement `delete(0, size());` and effectively change it to `delete(10, 90);` to delete only the characters inside the narrowed area.

The `narrow_position()` subroutine returns its argument p, adjusted so that it's inside the narrowed buffer boundaries.

Writing the buffer to a file ignores narrowing. Reading a file into the buffer lifts any narrowing in effect by setting `narrow_start` and `narrow_end` to zero.

8.1.6 Undo

```
int undo_op(int is_undo)
undo_mainloop()
undo_redisplay()
user buffer int undo_size;
```

With a nonzero argument, the `undo_op()` primitive undoes one basic operation like the undo command, described on page 83. With an argument of zero, it acts like redo. It returns a bit pattern describing what types of operations were undone or redone. The bit codes are defined in codes.h.

`UNDO_INSERT` means that originally an insertion occurred, and it was either undone or redone. The `UNDO_DELETE` and `UNDO_REPLACE` codes are similar.

Epsilon groups individual buffer changes into groups, and undoes one group at a time. While saving changes for undoing, Epsilon begins a new group when it redisplays buffers or when it begins a new command in the main loop. The `UNDO_REDISP` code indicates the former happened, and `UNDO_MAINLOOP` the latter. `UNDO_MOVE` indicates movement is being undone, and `UNDO_END` is used when Epsilon could only undo part of a command. If `undo_op()` returns zero, the buffer was not collecting undo information (see below).

Epsilon automatically starts a new undo group each time it does normal redisplay or passes through its main loop, by calling either the `undo_redisplay()` or `undo_mainloop()` primitives, respectively. You can call either of these primitives yourself to make Epsilon start a new undo group.

In addition to starting a new group, the `undo_mainloop()` primitive also makes the current buffer start to collect undo information. When you first create a buffer, Epsilon doesn't keep undo information for it, so that "system" buffers don't have this unnecessary overhead. Each time it passes through the main loop, Epsilon calls `undo_mainloop()`, and this makes the current buffer start collecting undo information, if it isn't already, and if the buffer-specific variable `undo_size` is nonzero.

```
int undo_count(int is_undo)
```

The `undo_count()` primitive takes a parameter that specifies whether undoing or redoing is involved, like `undo_op()`. The primitive returns a value indicating how much undoing or redoing information is saved. The number doesn't correspond to a particular number of commands, but to their complexity.

```
user buffer int undo_flag;
```

In addition to buffer changes and movements, Epsilon can record other information in its list of undoable operations. Each time you set the `undo_flag` variable, Epsilon inserts a "flag" in its undo list with the particular value you specify. When Epsilon is undoing or redoing and encounters a flag, it immediately ends the current group of undo operations and returns a code with the `UNDO_FLAG` bit on. It puts the value of the flag it encountered in the `undo_flag` variable. The yank-pop command uses flags 1 and 2 for undoing the previous yank.

8.1.7 Searching Primitives

```
user int matchstart;
user int matchend;
int search(int dir, char *str)
user short abort_searching;
#define ABORT_JUMP     -1
#define ABORT_ERROR    -2
```

The search primitives each look for the first occurrence of some text in a particular direction from point. Use 1 to specify forward, −1 to specify backward. They move point to the far end of the match, and set the `matchstart` and `matchend` variables to the near and far ends of the match, respectively. For example, if the buffer contains "abcd" and you search backward from the end for "bc", point and `matchend` will be 1 (between the 'a' and the 'b') and `matchstart` will be 3. If the search text does not appear in the buffer, point goes to the appropriate end of the buffer. These primitives return 1 if they find the text and 0 if not.

The most basic searching function is the `search()` primitive. It takes a direction and a string, and searches for the string. It returns 1 if it finds the text, or 0 if it does not.

If the user presses the abort key during searching, Epsilon's behavior depends upon the value of the `abort_searching` variable. If it's `0`, the key is ignored and the search continues. If it's `ABORT_JUMP` (the default), Epsilon aborts the search and jumps by calling the `check_abort()` primitive. If it's `ABORT_ERROR`, Epsilon aborts the search and returns the value `ABORT_ERROR`. The `search()`, `re_search()`, `re_match()`, and `buffer_sort()` primitives all use the `abort_searching` variable to control aborting.

```
user buffer short case_fold;
```

If the `case-fold` buffer-specific variable is nonzero, characters that match except for case count as a match. Otherwise, only exact matches (including case) count. To alter folding rules, see page 287.

Regular Expression Searching

```
int re_search(int flags, char *pat)
int re_compile(int flags, char *pat)
int re_match()
#define RE_FORWARD      0
#define RE_REVERSE      2
#define RE_FIRST_END    4
#define RE_SHORTEST     8
#define RE_IGNORE_COLOR 16
```

Several searching primitives deal with a powerful kind of pattern known as a *regular expression*. Regular expressions allow you to search for complex patterns. Regular expressions are strings formed according to the rules on page 55.

The `re_search()` primitive searches the buffer for one of these patterns. It operates like the `search()` primitive, taking a direction and pattern and returning `1` if it finds the pattern. It moves to the far end of the pattern from the starting point, and sets `matchstart` to the near end. If it doesn't find the pattern, or if the pattern is illegal, it returns `0`. In the latter case point doesn't move, in the former point moves to the end (or beginning) of the buffer.

When you specify a direction using `1` or `-1`, Epsilon selects the first-ending, longest match, unless the search string overrides this. However, instead of providing a direction (`1` or `-1`) as the first parameter to `re_search()` or `re_compile()`, you can provide a set of flags. These let you specify finding the shortest possible match, for example, without altering the search string.

The `RE_FORWARD` flag searches forward, while the `RE_REVERSE` flag searches backward. (If you don't include either, Epsilon searches forward.) The `RE_FIRST_END` flag says to find a match that ends first, rather than one that begins first. The `RE_SHORTEST` flag says to find the shortest possible match, rather than the longest. However, if the search string contains sequences that specify first-ending, first-beginning, shortest, or longest matches, those sequences override any flags.

A pattern may include color class assertions, as described on page 63. The `RE_IGNORE_COLOR` flag makes Epsilon ignore such assertions. The `do_color_searching()` subroutine uses this; if your search might include such assertions, calling that subroutine instead of these primitives will take care of ensuring that the buffer's syntax highlighting is up to date.

The `re_compile()` primitive checks a pattern for legality. It takes the same arguments as `re_search()` and returns `1` if the pattern is illegal, otherwise `0`. The `re_match()` primitive tells if the last-compiled pattern matches at this location in the buffer, returning the far end of the match if it does, or `-1` if it does not.

```
int parse_string(int flags, char *pat, ?char *dest)
int matches_at(int pos, int dir, char *pat)
int matches_at_length(int pos, int dir, char *pat)
int matches_in(int start, int end, char *pat)
```

The `parse_string()` primitive looks for a match starting at point, using the same rules as `re_match()`. It takes a direction (or flags) and a pattern like `re_compile()`, and a character pointer. It looks for a match of the pattern beginning at point, and returns the length of such a match, or zero if there was no match.

The third argument `dest` may be a null pointer, or may be omitted entirely. But if it's a pointer to a character array, `parse_string()` copies the characters of the match there, and moves point past them. If the pattern does not match, `dest` isn't modified.

The `matches_at()` subroutine accepts a regular expression `pat` and returns nonzero if the given pattern matches at a particular position in the buffer in the given direction. The `matches_at_length()` subroutine is similar, but it returns the length of the match, or zero if there was no match.

The `matches_in()` subroutine accepts a regular expression `pat` and searches for the pattern in the specified buffer range, returning nonzero to indicate it matches. Neither `matches_at()` nor `matches_in()` move point.

```
int find_group(int n, int open)
```

The `find_group()` primitive tells where in the buffer certain parts of the last pattern matched. It counts opening parentheses used for grouping in the last pattern, numbered from 1, and returns the position it was at when it reached a certain parenthesis. If `open` is nonzero, it returns the position of the n'th left parenthesis, otherwise it returns the position of its matching right parenthesis. If `n` is zero, it returns information on the whole pattern. If `n` is too large, or negative, the primitive aborts with an error message.

Searching Subroutines

```
int do_searching(int flags, char *str)  /* search.e */
int do_color_searching(int flags, char *str)  /* search.e */
```

The `do_searching()` subroutine defined in search.e is handy when you want to use a variable to determine the type of search. A `flags` value of `0` means perform a plain forward search. The flags `REVERSE`, `REGEX`, and `WORD` specify a reverse search, a regular expression search, or a word search, respectively. The subroutine normally performs case-folding if the buffer's `case_fold` variable is non-zero; pass `MODFOLD` to force Epsilon to search without case-folding, or pass `MODFOLD` and `FOLD` to force Epsilon to case-fold. The above flags may be combined in any combination.

The `do_searching()` subroutine returns 1 on a successful search, or `0` if the search text was not found. It can also return `DSABORT` if the user aborted the search (see the `abort_searching` variable) or `DSBAD` if the (regular expression) search pattern was invalid. If the search was successful, Epsilon moves to just after the found text (or just before, for reverse searches); in all other cases point doesn't change.

The `do_color_searching()` subroutine defined in search.e takes parameters and returns values just like `do_searching()`, but it handles regular expressions that use assertions like `<c:perl-comment>` to match based on the colors applied via syntax highlighting. If you use such syntax in a primitive like `re_search()`, Epsilon will search based on the syntax highlighting currently applied to the buffer. Because Epsilon computes syntax highlighting only as needed during screen display, as well as in the background, the buffer's syntax highlighting may not be up to date. This subroutine ensures that the buffer's syntax highlighting is up to date as it finds matches, by reparsing and recoloring the buffer whenever it has to.

```
int word_search(int dir, char *str)
int narrowed_search(int flags, char *str, int limit)
```

If `do_searching()` needs to search in word mode, it calls the `word_search()` subroutine. This function searches for `str`, rejecting matches unless they are preceded and followed by non-word characters. More precisely, it converts the text into a regular expression pattern, constructed so that each space in the original pattern matches any sequence of whitespace characters, and each word in the pattern only matches whole words.

The `narrowed_search()` subroutine is like `do_searching()`, but takes a parameter `limit` to limit the search. Epsilon will only search a region of the buffer within `limit` characters of its starting point. For example, if point is at 30000 and you call `narrowed_search()` and specify a reverse search with a limit of 1000, the match must occur between positions 29000 and 30000. If no such match is found, point will be set to 29000 and the function will return 0.

```
string_replace(char *str, char *with, int flags)
show_replace(char *str, char *with, int flags)
```

The `string_replace()` subroutine allows you to do string replacements from within a function. It accepts flags from the same list as `do_searching()`. Provide the `INCR` flag if you want the subroutine to display the number of matches it found, and the number that were replaced. Provide the `QUERY` flag to ask the user to confirm each replacement. This subroutine sets the variables `replace-num-found` and `replace-num-changed` to indicate the total number of replacements it found, and the number the user elected to change.

If you want to display what will be replaced without replacing anything, call the `show_replace()` subroutine. It takes the same parameters as `string_replace()`, and displays a message in the echo area. All Epsilon's replacing commands call this subroutine to display their messages.

```
simple_re_replace(int dir, char *str, char *repl)
```

The `simple_re_replace()` subroutine performs a regular expression replacement on the current buffer. It searches through the buffer, starting from the top, and passing `dir` and `str` directly to the `re_search()` primitive. It deletes each match and inserts the string `repl` instead. The replacement text is inserted literally, with no interpolation. If you want to use #1 in your replacement text, get a count of the matches, or other more involved things, call `string_replace()` instead.

```
int search_read(char *str, char *prmpt, int flags)
int default_fold(int flags)
int get_search_string(char *pr, int flags)
char *default_search_string(int flags)
```

To ask the user for a search string, use the `search_read()` subroutine. Its parameter `str` provides an initial search string, and it returns a set of flags which you can pass to `do_searching()`. It takes an initial set of flags, which you can use to start the user in one of the searching modes. Call `default_fold()` with any flags before calling `search_read()`. It will turn on any needed flags relating to case-folding, based on the value of the `case_fold` variable, and return a modified set of flags.

The function leaves the string in either the `_default_search` or the `_default_regex_search` variable, depending upon the searching flags it returns. You can call the `default_search_string()` subroutine with that set of searching flags and it will return a pointer to the appropriate one of these. Depending on what the user types, the `search_read()` subroutine may perform searching itself, in addition to returning the search string.

The `get_search_string()` subroutine asks the user for a string to search for by calling `search_read()`.

```
buffer int (*search_continuation)();
int sample_search_continuation(int code, int flags, char *str)
```

In some modes a buffer may contain a single "record" out of many. Records may be swapped by changing the narrowing on the buffer (as in Info mode), while in other modes the contents of the buffer may be completely replaced with text from a different record.

A mode may wish to let users search from one record to the next, when no more matches can be found in the current record. (This capability relates to searching by the user, with the `search_read()` subroutine, not the primitive searching functions.)

A mode may set the buffer-specific `search_continuation` function pointer to a search-continuation function if it wants this behavior. If it's nonzero, the searching functions will call this function to advance to a different record, or to remember or return to a particular record.

Epsilon assumes that the set of possible records have an implicit order to them, forming a list. And it assumes that a record id, referring to a specific record, may be stored in a character array of length `FNAMELEN`.

The `code` parameter indicates the desired operation. If `SCON_RECORD`, the search-continuation function must write a record id for the current record into the array `str`. If `SCON_RESTORE`, it must return to the record identified by the previously-saved id `str`. These operations should return zero. If `SCON_COMPARE`, it must compare the current record with the id saved in `str` (according to the record order), returning -1, 0, or 1 depending on whether the current record is before, equal to, or after the saved record, respectively.

Any other `code` means to move to the next or previous record, according to whether the `flags` parameter contains the `REVERSE` bit, and position to its start (or, for reverse searching, end). In this case, `code` becomes a count, starting from 1, that indicates the number of record positionings done since the last user keypress (for use in displaying progress messages). It should return 1 on success, or 0 if there were no more records (and should remain at the original record in that case).

A search-continuation function may wish to pre-screen records, and skip over those that do not contain the search string (but is not required to do so). If it chooses to do this, it can use `flags` and `str` to call the `do_searching()` subroutine; these specify the search being performed.

```
int col_search(char *str, int col)    /* search.e */
```

The `col_search()` subroutine defined in search.e attempts to go to the beginning of the next line containing a certain string starting in a certain column. It returns 1 if the search is successful, 0 otherwise.

```
int line_search(int dir, char *s)     /* grep.e */
int prox_line_search(char *s)     /* tags.e */
```

The `line_search()` subroutine searches in direction `dir` for a line containing only the text `s`. It returns 1 if found, otherwise 0.

The `prox_line_search()` subroutine searches in the buffer for lines containing exactly the text `s`. It goes to the start of the closest such line to point, and returns 1. If there is no matching line, it returns 0.

```
do_drop_matching_lines(int flags, char *pat, int drop)
```

The `do_drop_matching_lines()` subroutine deletes all lines after point in the current buffer but those that contain the specified search pattern. The search flags say how to interpret the pattern. If `drop` is nonzero, the subroutine deletes lines that contain the pattern; if `drop` is zero it deletes all lines except those that contain the pattern. Temporarily set the `show-status` variable to zero to keep it from displaying a line count summary.

```
replace_in_readonly_hook(int old_readonly)
replace_in_existing_hook(int old_readonly)
```

The file-query-replace command calls some hook functions as it goes through its list of buffers or files. Just before it makes its first change in each buffer (or asks the user whether to make the change, if it's still in query mode), it calls either the `replace_in_existing_hook()` subroutine (if the buffer or file was already loaded before running the command) or the `replace_in_readonly_hook()` (if file-query-replace had to read the file itself). The file-query-replace command temporarily zeroes the `readonly-warning` variable; it passes the original value of this variable as a parameter to each hook.

The default version of `replace_in_existing_hook()` does nothing. The default version of `replace_in_readonly_hook()` warns about the file being read-only by calling `do_readonly_warning()`.

8.1.8 Moving by Lines

```
int nl_forward()
int nl_reverse()
int move_by_lines(int cnt)
```

The `nl_forward()` and `nl_reverse()` primitives quickly search for newline characters in the direction you specify. The `nl_forward()` primitive is the same as `search(1, "\n")`, while `nl_reverse()` is the same as `search(-1, "\n")`, where `\ n` means the newline character (see page 159). These primitives do not set `matchstart` or `matchend`, but otherwise work the same as the previous searching primitives, returning 1 if they find a newline and 0 if they don't.

The `move_by_lines()` primitive moves forward over `cnt` lines, like calling `nl_forward()` that many times. If `cnt` is negative, it moves backward, like calling `nl_reverse()`. It returns 0, unless it hit the end of the buffer (or narrowing) before moving the full amount; in that case it returns the number of lines still to go when it stopped. For example, if there are two newlines in the buffer before point, calling `move_by_lines(-10)` moves to the start of the buffer and returns –8.

```
to_begin_line()          /* eel.h macro */
to_end_line()            /* eel.h macro */
int give_begin_line()      /* basic.e */
int give_end_line()        /* basic.e */
```

The eel.h file defines textual macros named `to_begin_line()` and `to_end_line()` that make it easy to go to the beginning or end of the current line. They simply search in the appropriate direction for a newline character and back up over it if the search succeeds.

The `give_begin_line()` subroutine returns the buffer position of the beginning of the current line, and the `give_end_line()` subroutine returns the position of its end. Neither moves point.

```
go_line(int num)               /* basic.e */
int lines_between(int from, int to, ?int abort_ok)
```

```
int count_lines_in_buf(int buf, int abortok)
int buf_position_to_line_number(int buf, int pos)
int all_blanks(int from, int to)    /* indent.e */
```

The EEL subroutine `go_line()` defined in basic.e uses the `move_by_lines()` primitive to go to a certain line in the buffer, counting from 1. `go_line(2)`, for example, goes to the beginning of the second line in the buffer.

The `lines_between()` primitive returns the number of newline characters in the part of the buffer between `from` and `to`. If `abort_ok` is nonzero, the user can abort from this primitive, otherwise Epsilon ignores the abort key.

The `buf_position_to_line_number()` subroutine returns the line number, counting from 1, of a particular position in the specified buffer.

The `count_lines_in_buf()` subroutine returns the number of newline characters in the buffer `buf`. If `abortok` is nonzero and the user press the abort key, the subroutine uses the `check_abort()` primitive to abort.

The `all_blanks()` subroutine returns 1 if the characters between `from` and `to` are all whitespace characters (space, tab, or newline), 0 otherwise.

8.1.9   Other Movement Functions

```
int move_level(int dir, char *findch,
               char *otherch, int show, int stop_on_key)
buffer int (*mode_move_level)();
int c_move_level(int dir, int stop_on_key)
int html_move_level(int dir, int stop_on_key)
int default_move_level(int dir, char *findch,
                       char *otherch)
```

Several subroutines move through text counting and matching various sorts of delimiters. The `move_level()` subroutine takes a direction `dir` which may be 1 or −1, and two sets of delimiters. The routine searches for any one of the characters in `findch`. Upon finding one, it continues searching in the same direction for the character in the same position in `otherch`, skipping over matched pairs of these characters in its search.

For example, if `findch` was `">])"` and `dir` was −1, `move_level()` would search backwards for one of these three characters. If it found a ')' first, it would then select the third character of `otherch`, which might be a '('. It would then continue searching for a '('. But if it found additional ')' characters before reaching that '(', it would need to find additional '(' characters before stopping.

The subroutine returns 1 to indicate that it found a match, and leaves point on the far side of the match (like commands such as forward-level). If no match can be found, the subroutine returns 0. Additionally, if its parameter `show` is nonzero, it displays an "Unmatched delimiter" message. When no characters in `findch` can be found in the specified direction, it sets point to the far end of the buffer and returns 1. If `stop_on_key` is nonzero, the subroutine will occasionally check for user key presses, and abort its search if the user has pressed a key. It returns −2 in this case and doesn't change point.

Certain modes define a replacement level matcher that understands more of the syntax of that mode's language. They do this by setting the buffer-specific function pointer variable `mode_move_level` to a function such as `c_move_level()`. The `move_level()` subroutine will call this function instead of doing its normal processing when this variable is nonzero in the current buffer.

Any such function will receive only `dir` and `stop_on_key` parameters. (It should already know which delimiters are significant in its language.) It should return the buffer position it reached (but not actually move there), if it found a pair of matched delimiters, or if it reached one end of the buffer without finding any suitable delimiters. If should return `-1` if it detected an unmatched delimiter, or `-2` if a keypress made it abort.

The `default_move_level()` function is what `move_level()` calls when no mode-specific function is available. It takes parameters like `move_level()`, and returns `-1` or a buffer position like `c_move_level()`. A mode-specific function may wish to call this function, specifying a set of delimiters suitable for that language. The `html_move_level()` subroutine, for example, does just that.

```
int give_position(int (*cmd)())
```

The `give_position()` subroutine runs the subroutine `cmd`, which (typically) moves to a new position in the buffer. The `give_position()` subroutine returns this new position, but restores point to its original value. For example, `give_position(forward_word)` returns the buffer position of the end of the current word. EEL requires that `cmd` be declared before you call it, via a line like `int cmd();`, unless it's defined in the same file, before the `give_position()` call.

8.1.10   Sorting Primitives

```
buffer_sort(char *newbuf, ?int col, int rev)
do_buffer_sort(char *newbuf, int col, int rev)
sort_another(char *buf, int col, int rev)
do_sort_region(int from, int to, int col, int rev)
char show_status;
```

The EEL primitive `buffer_sort()` sorts the lines of the current buffer alphabetically. It does not modify the buffer, but rather inserts a sorted copy into the named buffer (which must be different). It performs each comparison starting at column `col`, which is optional and defaults to 0 (the first column). If the `case_fold` variable is nonzero, sorting ignores the case of letters. It sorts lines in reverse order if the optional `rev` parameter is 1, not 0. (To alter folding rules, see page 287.)

If the variable `show_status` is nonzero, Epsilon will display progress messages as the sort progresses. Otherwise, no status messages appear.

The `do_buffer_sort()` subroutine is similar, but respects the `sort-case-fold` variable, not `case-fold` like `buffer_sort()`.

The `sort_another()` subroutine takes the name of a buffer and sorts it in place. The parameter `col` specifies the column to sort on, and `rev`, if nonzero, requests a reverse sort.

The `do_sort_region()` subroutine sorts a portion of the current buffer in place. The `from` and `to` parameters specify the region to sort. The `col` parameter specifies the column to sort on, and the `rev` parameter, if nonzero, requests a reverse sort.

If the user presses the abort key during sorting, Epsilon's behavior depends upon the value of the `abort_searching` variable. If 0, the key is ignored and the sort will run to completion. If `ABORT_JUMP`, Epsilon aborts the sort and jumps by calling the `check_abort()` primitive. If `ABORT_ERROR`, Epsilon aborts the sort and returns `ABORT_ERROR`. Whenever Epsilon aborts a sort, nothing gets inserted in the newbuf buffer. (For the subroutines that sort in place, the buffer is not changed.) Except when aborted, the `buffer_sort()` primitive and all the sorting subroutines described above return 0.

8.1.11 Other Formatting Functions

```
right_align_columns(char *pat)
```

The `right_align_columns()` subroutine locates all lines containing a match for the regular expression pattern `pat`. It notes the ending column of each match. (It assumes that `pat` occurs no more than one per line.)

Then, if some matches end at an earlier column than others, it adds indentation before each match as needed, so all matches will end at the same column.

```
columnize_buffer_text(int buf, int width, int margin)
```

The `columnize_buffer_text()` subroutine takes the lines in the buffer `buf` and reformats them into columns. It leaves a margin between columns of `margin` spaces, and chooses the number of columns so that the resulting buffer is at most `width` characters wide (unless an original line in the buffer is already wider than `width`).

```
do_buffer_to_hex(char *b, char transp[256], ?int flags)
```

The `do_buffer_to_hex()` primitive writes a hex view of the current buffer to the buffer `b`, creating or emptying it first. It ignores any narrowing in the original buffer. It uses the 256 character `transp` array to help construct the last column of the hex view; each character from the buffer will be replaced by the character at that offset in the `transp` array. If the buffer contains Unicode characters with codes higher than 255, they'll appear as-is.

If a buffer might contain Unicode characters, the primitive uses a display format that leaves room for 16 bits per character; otherwise it uses a format with room for 8 bits per character. The optional `flags` argument, if 1, forces 8 bits per character. If any character in the buffer doesn't fit in 8 bits, only its lower 8 bits will be shown in the hex listing.

8.1.12 Comparing

```
int compare_buffer_text(int buf1, int pos1,
                        int buf2, int pos2, int fold)
int buffers_identical(int a, int b)
```

The `compare_buffer_text()` primitive compares two buffers, specified by buffer numbers, starting at the given offsets within each. If `fold` is nonzero, Epsilon performs case-folding as in searching before comparing each character, using the case-folding rules of the current buffer. The primitive returns the number of characters that matched before the first mismatch.

The `buffers_identical()` subroutine checks to see if two buffers, specified by their buffer numbers, are identical. It returns nonzero if the buffers are identical, zero if they differ. If neither buffer exists, they're considered identical; if one exists, they're different.

```
do_uniq(int incl_uniq, int incl_dups, int talk)
buf_sort_and_uniq(int buf)
```

The `do_uniq()` subroutine defined in uniq.e goes through the current buffer comparing each line to the next, and deleting each line unless it meets certain conditions.

If `incl_uniq` is nonzero, lines that aren't immediately followed by an identical line will be preserved. If `incl_dups` is nonzero, the first copy of each line that is immediately followed by one or more identical lines will be preserved. (The duplicate lines that follow will always be deleted.)

If `talk` is nonzero, the subroutine will display status messages as it proceeds.

The `buf_sort_and_uniq()` subroutine sorts the specified buffer and discards duplicate lines in it, with no status messages.

```
do_compare_sorted(int b1, int b2, char *only1,
                  char *only2, char *both)
```

The `do_compare_sorted()` subroutine works like the compare-sorted-windows command, but lets you specify the two buffers to compare, and the names of the three result buffers. Any of the result buffer names may be NULL, and the subroutine won't generate data for that buffer.

```
int tokenize_lines(int buf1, int **lines1, int *len1,
                   int buf2, int **lines2, int *len2)
int lcs(int *lines1, int len1, int *lines2, int len2, char *outbuf)
```

These primitives help to compute a minimum set of differences between the lines of two buffers `buf1` and `buf2`. See the implementation of the diff command for an example of their use.

Call the `tokenize_lines()` primitive first. It begins by counting the lines in each buffer (placing the results in `len1` and `len2`). Then it uses the `realloc()` primitive to make room in the arrays passed by reference as `lines1` and `lines2`, which may be null at the start. Each array will have room for one token (unique integer) for each line of its buffer. (The arrays may be freed after calling `lcs()`, or reused in later calls.)

The `tokenize_lines()` primitive then fills in the arrays with unique tokens, chosen so that two lines will have the same token if and only if they're identical.

The `lcs()` primitive takes the resulting arrays and line counts, and writes a list of shared line ranges to the specified buffer, one per line, in ascending order. Each line range consists of a line number for the first buffer, a line number for the second (both 0-based) and a line count. For instance, a line "49 42 7" indicates that the seven lines starting at line 49 in the first buffer match the seven lines starting at line 42 in the second (counting lines from 0).

```
int lcs_char(int buf1, int from1, int to1,
             int buf2, int from2, int to2, char *outbuf)
```

The `lcs_char()` primitive is a character-oriented version of the `tokenize_lines()` and `lcs()` primitives described above. It compares ranges of characters in a pair of buffers.

It writes a list of shared character ranges to the specified buffer, one per line, in ascending order. Each character range consists of a character offset for the first buffer relative to `from1`, a character offset for the second buffer relative to `from2`, and a character count. For instance, a line "49 42 7" in the output buffer indicates that the seven characters in the range `from1 + 47` to `from1 + 47 + 7` in the first buffer match the seven characters in the range `from2 + 42` to `from2 + 42 + 7` in the second.

8.1.13 Managing Buffers

```
int create(char *buf)
char *bufnum_to_name(int bnum)
int name_to_bufnum(char *bname)
int zap(char *buf)
buf_zap(int bnum)
int change_buffer_name(char *newname)
```

The `create()` primitive makes a new buffer. It takes the name of the buffer to create. If the buffer already exists, nothing happens. In either case, it returns the buffer number of the buffer.

Some primitives let you specify a buffer by name; others let you specify a buffer by number. Epsilon tries never to reuse buffer numbers, so EEL functions can look a buffer up by its buffer number to see if a particular buffer still exists. Functions that accept a buffer number generally start with `buf_`.

Use the `bufnum_to_name()` primitive to convert from a buffer number to the buffer's name. If no such buffer exists, it returns a null pointer. The `name_to_bufnum()` primitive takes a buffer name, and gives you the corresponding buffer number. If no such buffer exists, it returns zero.

The `zap()` primitive creates a buffer if necessary, but empties it of all characters if the buffer already exists. So calling `zap()` always results in an empty buffer. The `zap()` primitive returns the buffer number of the buffer, whether or not it needed to create the buffer. The `buf_zap()` primitive works like `zap()`, except the former takes a buffer number instead of a buffer name, and signals an error if no buffer with that number exists. Unlike `zap()`, `buf_zap()` cannot create a buffer. Neither primitive switches to the emptied buffer.

The `change_buffer_name()` primitive renames the current buffer to the indicated name. If there is already a buffer with the new name, the primitive returns 0, otherwise the buffer is renamed and the primitive returns 1.

```
int exist(char *buf)
int buf_exist(int bnum)
delete_buffer(char *buf)
delete_user_buffer(char *buf)
buf_delete(int bnum)
drop_buffer(char *buf)          /* buffer.e */
char *temp_buf()        /* basic.e */
int tmp_buf()       /* basic.e */
```

The `exist()` primitive tells whether a buffer with a particular name exists. It returns 1 if the buffer exists, 0 if not. The `buf_exist()` does the same thing, but takes a buffer number instead of a buffer name.

The `delete_buffer()` primitive removes a buffer with a given name. It also removes all windows associated with the buffer. The `buf_delete()` primitive does the same thing, but takes a buffer number. Epsilon signals an error if the buffer does not exist, if it contains a running process, or if one of the buffer's windows could not be deleted. If the buffer might have syntax highlighting in it, use the `delete_user_buffer()` subroutine instead; it cleans up some data needed by syntax highlighting.

The `drop_buffer()` subroutine deletes the buffer, but queries the user first like the kill-buffer command if the buffer contains unsaved changes.

The EEL subroutine `temp_buf()`, defined in basic.e, uses the `exist()` primitive to create an unused name for a temporary buffer. It returns the name of the empty buffer it creates. The `tmp_buf()` subroutine creates a temporary buffer like `temp_buf()`, but returns its number instead of its name.

```
buffer char *bufname;
buffer int bufnum;
```

The `bufname` variable returns the name of the current buffer, and the `bufnum` variable gives its number. Setting either switches to a different buffer. If the indicated buffer does not exist, nothing happens. Use this method of switching buffers only to temporarily switch to a new buffer; use the `to_buffer()` or `to_buffer_num()` subroutines described on page 223 to change the buffer a window will display.

To set the `bufname` variable, use the syntax `bufname` = *new value*;. Don't use `strcpy()`, for example, to modify it.

```
int buffer_size(char *buf)
int buf_size(int bnum)
int get_buf_point(int buf)
set_buf_point(int buf, int pos)
```

The `buffer_size()` and `buf_size()` subroutines returns the size in characters of the indicated buffer (specified by its name or number). The `get_buf_point()` subroutine returns the value of point in the indicated buffer. The `set_buf_point()` subroutine sets point in the specified buffer to the value `pos`. These are all defined in buffer.e.

8.1.14   Catching Buffer Changes

```
user buffer short call_on_modify;
on_modify()      /* buffer.e */
zeroed buffer (*buffer_on_modify)();
buffer char _buf_readonly;
check_modify(int buf)
```

If the buffer-specific `call_on_modify` variable has a nonzero value in a particular buffer, whenever any primitive tries to modify that buffer, Epsilon calls the EEL subroutine `on_modify()` first. By default, that subroutine calls the `normal_on_modify()` subroutine, which aborts the modification if the buffer-specific variable `_buf_readonly` is nonzero, indicating a read-only buffer, and does various similar things.

But if the `buffer_on_modify` buffer-specific function pointer is nonzero for that buffer, `on_modify()` instead calls the subroutine it indicates. That subroutine may wish to call `normal_on_modify()` itself.

An `on_modify()` function can abort the modification or set variables. But if it plans to return, it must not create or delete buffers, or permanently switch buffers.

One of `normal_on_modify()`'s tasks is to handle read-only buffers. There are several types of these, distinguished by the value of the `_buf_readonly` variable, which if nonzero indicates the buffer is read-only. A value of 1 means the user explicitly set the buffer read-only. The value 2 means Epsilon automatically set the buffer read-only because its corresponding file was read-only.

A value of 3 indicates pager mode; this is just like a normal read-only buffer, but if the user action causing the attempt at buffer modification happens to be the result of the ⟨Space⟩ or ⟨Backspace⟩ keys, Epsilon cancels the modification and pages forward or backward, respectively.

The `check_modify()` primitive runs the `on_modify()` function on a specified buffer (if `call_on_modify` is nonzero in that buffer). You can use this if you plan to modify a buffer later but want any side effects to happen now. If the buffer is marked read-only, this function will abort with an error message. If the buffer is in virtual mode and its cursor is positioned in virtual space, Epsilon will insert

whitespace characters to reach the virtual column. Because this can change the value of point, you should call `check_modify()` before passing the values of spots to any function.

For example, suppose you write a subroutine to replace the previous character with a '+', using a statement like `replace(point - 1, '+');`. Suppose point has the value 10, and appears at the end of a line containing 'abc' (in column 3). Using virtual mode, the user might have positioned the cursor to column 50, however. If you used the above statement, Epsilon would call `replace()` with the value 9. Before replacing, Epsilon would call `on_modify()`, which, in virtual mode, would insert tabs and spaces to reach column 50, and move point to the end of the inserted text. Then Epsilon would replace the character 'c' at buffer position 9 with '+'. If you call `check_modify(bufnum);` first, however, Epsilon inserts its tabs and spaces to reach column 50, and `point - 1` correctly refers to the last space it inserted.

```
reset_modified_buffer_region(char *tag)
int modified_buffer_region(int *from, int *to, ?char *tag)
```

Sometimes an EEL function needs to know if a buffer has been modified since the last time it checked. Epsilon can maintain this information using tagged buffer modification regions.

An EEL function first tells Epsilon to begin collecting this information for the current buffer by calling the `reset_modified_buffer_region()` primitive and passing a unique tag name. (Epsilon's syntax highlighting uses a modified buffer region named `needs-color`, for instance.) Later it can call the `modified_buffer_region()` primitive, passing the same tag name. Epsilon will set its `from` and `to` parameters to indicate the range of the buffer that has been modified since the first call.

For example, say a buffer contains six characters abcdef when `reset_modified_buffer_region()` is called. Then the user inserts and deletes some characters resulting in abxyf. A `modified_buffer_region()` would now report that characters in the range 2 to 4 have been changed. If the buffer contains many disjoint changes, `from` will indicate the start of the first change, and `to` the end of the last.

The `modified_buffer_region()` primitive returns 0 if the buffer hasn't been modified since the last `reset_modified_buffer_region()` with that tag. In this case `from` and `to` will be equal. (They might also be equal if only deletion of text had occurred, but then the primitive wouldn't have returned 0.) It returns 1 if the buffer has been modified. If `reset_modified_buffer_region()` has never been used with the specified tag in the current buffer, it returns −1, and sets the `from` and `to` variables to indicate the whole buffer.

The `tag` may be omitted when calling `modified_buffer_region()`. In that case Epsilon uses an internal tag that's reset on each buffer display. So the primitive indicates which part of the current buffer has been modified since the last buffer display.

8.1.15 Listing Buffers

```
char *buffer_list(int start)
int buf_list(int offset, int mode)
```

The `buffer_list()` primitive gets the name of each buffer in turn. Each time you call this primitive, it returns the name of another buffer. It begins again when given a nonzero argument. When it has returned the names of all the buffers since the last call with a nonzero argument, it returns a null pointer.

The `buf_list()` primitive can return the number of each existing buffer, one at a time, like `buffer_list()`. The `mode` can be 0, 1, or 2, to position to the lowest-numbered buffer in the list, the last buffer returned by `buf_list()`, or the highest-numbered buffer, respectively. The `offset` lets you advance

from these buffers to lower or higher-numbered buffers, by providing a negative or positive offset. Unlike `buffer_list()`, this primitive lets you back up or go through the list backwards.

For example, this code fragment displays the names of all buffers, one at a time, once forward and once backward:

```
s = buffer_list(1);
do {
        say("Forward %d: %s", name_to_bufnum(s), s);
} while (s = buffer_list(0));

i = buf_list(0, 2);
do {
        say("Back %d: %s", i, bufnum_to_name(i));
} while (i = buf_list(-1, 1));
say("Done.");
```

8.2   Display Primitives

8.2.1   Creating & Destroying Windows

```
window_kill()
window_one()
```

The `window_kill()` primitive removes the current window if possible, in the same way as the kill-window command does. The `window_one()` primitive eliminates all but the current window, as the command one-window does.

```
remove_window(int win)
```

The `remove_window()` primitive deletes a window by handle or number. If you delete a tiled window, Epsilon expands other windows as needed to fill its space. You cannot delete the last remaining tiled window.

```
int give_window_space(int dir)
#define BLEFT   0   /* direction codes */
#define BTOP    1
#define BRIGHT  2
#define BBOTTOM 3
```

The `give_window_space()` primitive deletes the current window. It expands adjacent windows in the specified direction into the newly available space, returning 0. If there are no windows in the specified direction, it does nothing and returns 1.

```
window_split(int orientation)
#define HORIZONTAL      (0)
#define VERTICAL        (1)
```

The `window_split()` primitive makes two windows from the current window, like the commands split-window and split-window-vertically do. The argument to `window_split()` tells whether to make the new windows appear one on top of the other (with argument `HORIZONTAL`) or side-by-side (with argument `VERTICAL`). The standard EEL header file, eel.h, defines the macros `HORIZONTAL` and `VERTICAL`. The primitive returns zero if it could not split the window, otherwise nonzero. When you split the window, Epsilon automatically remembers to call the `prepare_windows()` and `build_mode()` subroutines during the next redisplay.

```
user short window_handle;
user short window_number;
next_user_window(int dir)
```

You may refer to a window in two ways: by its *window handle* or by its *window number*.

Epsilon assigns a unique window handle to a window when it creates the window. This window handle stays with the window for the duration of that window's lifetime. To get the window handle of the current window, use the `window_handle` primitive.

The window number, on the other hand, denotes the window's current position in the window order. You can think of the window order as the position of a window in a list of windows. Initially the list has only one window. When you split a window, the two child windows replace it in the list. The top or left window comes before the bottom or right window. When you delete a window, that window leaves the list. The window in the upper left has window number `0`. Pop-up windows always come after tiled windows in this order, with the most recently created (and therefore topmost) pop-up window last. The `window_number` primitive gives the window number of the current window.

Epsilon treats windows in a dialog much like pop-up windows, assigning each a window number and window handle. The stacking order of dialogs is independent of their window handles, however. Deleting all the windows on a dialog makes Epsilon remove the dialog. (Epsilon doesn't count windows with the `system_window` flag set when determining if you've deleted the last window.)

To change to a different window, you can set either the `window_handle` or `window_number` variables. Epsilon then makes the indicated window become the current window. Epsilon interprets `window_number` modulo the number of windows, so window number `-1` refers to the last window.

Many primitives that require you to specify a window will accept either its handle or its number. Use `window_handle` to remember a particular window, since its number can change as you add or delete windows.

You can increment or decrement the `window_number` variable to cycle through the list of available windows. But it's usually better to use the `next_user_window()` subroutine, passing it `1` to go to the next window or `-1` to go to the previous one. This will skip over system windows.

```
int number_of_windows()
int number_of_popups()
int number_of_user_windows()
int is_window(int win)
#define ISTILED        1
#define ISPOPUP        2
```

The `number_of_windows()` primitive returns the total number of windows, and the `number_of_popups()` primitive returns the number of pop-up windows. The `number_of_user_windows()` subroutine returns the total number of windows, excluding system windows.

The `is_window()` primitive accepts a window handle. It returns `ISTILED` if the value refers to a conventional tiled window, `ISPOPUP` if the value refers to a pop-up window or a window in a dialog, or `0` if

the value does not refer to a window. Unlike most window functions, it accepts only a window handle, not a window number.

8.2.2 Window Resizing Primitives

```
user window short window_height;
user window short window_width;
int text_height()
int text_width()
```

The `window_height` variable contains the height of the current window in lines, including any mode line or borders. Setting it changes the size of the window. Each window must have at least one line of height. The `window_width` variable contains the width of the current window, counting any borders the window may have. If you set these variables to illegal values, Epsilon will adjust them to the closest legal values.

The `text_height()` and `text_width()` primitives, on the other hand, exclude borders and mode lines from their calculations, returning only the number of lines or columns of the window available for the display of text.

```
int window_edge(int orien, int botright)
#define TOPLEFT        (0)
#define BOTTOMRIGHT    (1)
```

The `window_edge()` primitive tells you where on the screen the current window appears. For the first parameter, specify either `HORIZONTAL` or `VERTICAL`, to get the column or row, respectively. For the second parameter, provide either `TOPLEFT` or `BOTTOMRIGHT`, to specify the corner. Counting starts at the upper left corner of the screen, which has `0` for both coordinates.

8.2.3 Preserving Window Arrangements

```
struct window_info {
        short left, top, right, bottom;
        short textcolor, hbordcolor;
        short vbordcolor, titlecolor;
        short borders, other, bufnum;
        int point, dpoint;
        /* primitives fill in before this line */
        int dcolumn;
        short prevbuf;
};

get_window_info(int win, struct window_info *p)
low_window_info(int win, struct window_info *p)
window_create(int first, struct window_info *p)
low_window_create(int first, struct window_info *p)
select_low_window(int wnum, int top, int bot,
                  int lines, int cols)
```

Epsilon has several primitives that are useful for recording a particular window configuration and reconstructing it later.

The `get_window_info()` subroutine fills a structure with information on the specified window. The information includes the window's size and position, its selected colors, and so forth. It uses the `low_window_info()` primitive to collect some of the information, then fills in the rest itself by inspecting the window.

After calling `get_window_info()` on each tiled window (obtaining a series of structures, each holding information on one window), you can restore that window configuration using the `window_create()` subroutine. It takes a pointer to a structure that `get_window_info()` filled in, and a flag that must be nonzero if this is the first window in the new configuration. It uses the `low_window_create()` primitive to create the window. The `point` or `dpoint` members of the structure may be −1 when you call `window_create()` or `low_window_create()`, and Epsilon will provide default values for `point` and `window_start` in the new window, based on values stored with the buffer. The window-creating functions remain in the window they create, so you can modify its window-specific variables.

After a series of `window_create()`'s, you must use the `select_low_window()` primitive to switch to one of the created windows (specifying it by window number or handle, as usual).

Using `window_create()` directly modifies windows, and Epsilon doesn't check that the resulting window configuration is legal. For example, you can define a set of tiled windows that leave gaps on the screen, overlap, or extend past the screen borders. The result of creating an illegal window configuration is undefined.

The first time you call `window_create()`, pass it a nonzero flag, and Epsilon will (internally) delete all tiled windows, and create the first window. Then call `window_create()` again, as needed, to create the remaining windows (pass it a zero flag). Finally, you must call the `select_low_window()` primitive. Once you begin using `window_create()`, Epsilon will not be able to refresh the screen correctly until you call the `select_low_window()` primitive to exit window-creation. The `top` and `bot` parameters specify the new values of the `avoid-top-lines` and `avoid-bottom-lines` variables, and set the variables to the indicated values while finishing window creation. The `lines` and `cols` parameters specify the size of the screen that was used to construct the old window configuration. All windows defined using `low_window_create()` are based on that screen size. When you call `select_low_window()`, Epsilon resizes all the windows you've defined so that they fit the current screen size.

```
save_screen(struct screen_info *p)
restore_screen(struct screen_info *p)
```

The `save_screen()` subroutine saves Epsilon's window configuration in a `struct screen_info` structure. The first time you call this subroutine on an instance of the `screen_info` structure, make sure its `wins` member is zero. The `restore_screen()` subroutine restores Epsilon's window configuration from such a structure.

8.2.4 Pop-up Windows

```
int add_popup(column, row, width, height, border, bnum)
   /* macros for defining a window's borders */
   /* BORD(BTOP, BSINGLE) puts single line on top */
#define BLEFT   0
#define BTOP    1
#define BRIGHT  2
#define BBOTTOM 3
#define BNONE   0
#define BBLANK  1
#define BSINGLE 2
```

```
#define BDOUBLE 3
#define BORD(side, val)        (((val) & 3) << ((side) * 2))
#define GET_BORD(side, bord)    ((bord >> (side * 2)) & 3)
#define LR_BORD(val)  (BORD(BLEFT, (val)) + BORD(BRIGHT, (val)))
#define TB_BORD(val)  (BORD(BTOP, (val)) + BORD(BBOTTOM, (val)))
#define ALL_BORD(val)  (LR_BORD(val) + TB_BORD(val))
```

The `add_popup()` primitive creates a new pop-up window. It accepts the `column` and `row` of the upper left corner of the new window, and the `width` and `height` of the window (including any borders). The `border` parameter contains a code saying what sort of borders the window should have, and the `bnum` parameter gives the buffer number of the buffer to display in the window. The primitive returns the handle of the new window, or −1 if the specified buffer did not exist, so Epsilon couldn't create the window. If the pop-up window is to become part of a dialog (see page 322), its size, position and border will be determined by the dialog, not the values passed to `add_popup()`.

You can define the borders of a window using macros from codes.h. For each of the four sides, you can specify no border, a blank border, a border drawn with a single line, or a border drawn with a double line, using the codes BNONE, BBLANK, BSINGLE, or BDOUBLE, respectively. Specify the side to receive the border with the macros BLEFT, BTOP, BRIGHT, and BBOTTOM. You can make a specification for a given side using the BORD() macro, writing BORD(BBOTTOM, BDOUBLE) to put a double-line border at the bottom of the window. Add the specifications for each side to get the complete border code.

You can use other macros to simplify the border specification. Write `LR_BORD(BSINGLE) + TB_BORD(BDOUBLE)` to produce a window with single-line borders on the left and right, and double-line borders above and below. Write `ALL_BORD(BNONE)` for a window with no borders at all, and the most room for text.

You can use the `GET_BORD()` macro to extract (from a complete border code) the specification for one of its sides. For example, to find the border code for the left-side border of a window with a border value of `bval`, write `GET_BORD(BLEFT, bval)`. If the window has a double-line border on that side, the macro would yield BDOUBLE.

```
int window_at_coords(int row, int col, ?int screen)
```

The `window_at_coords()` primitive provides the handle of the topmost window at a given set of screen coordinates. The primitive returns −1 if no window occupies that part of the screen. The screen number parameter can be zero or omitted to refer to the main screen, but it is usually a screen number from the `mouse_screen` primitive.

```
int window_to_screen(int win)
```

The `window_to_screen()` primitive takes a window handle and returns its screen number. Windows that are part of a dialog box have nonzero screen numbers; in this version other windows always have a screen number of zero.

```
int screen_to_window(int screen)
```

The `screen_to_window()` primitive takes a screen number, as returned in the variable `mouse_screen`, and returns the window handle associated with it. If the screen number is zero, there may be several windows associated with it; Epsilon will choose the first one. In this version of Epsilon, nonzero screen numbers uniquely specify a window.

```
user window int window_left;
user window int window_top;
```

The `window_left` and `window_top` primitive variables provide screen coordinates for the current window. You can set the coordinates of a pop-up window to move the window around. Epsilon ignores attempts to set these variables in tiled windows.

8.2.5 Pop-up Window Subroutines

```
view_buffer(char *buf, int last)   /* complete.e */
view_buf(int buf, int last)        /* complete.e */
```

Several commands in Epsilon display information using the `view_buffer()` subroutine. It takes the name of a buffer and displays it page by page in a pop-up window. The `view_buf()` subroutine takes a buffer number and does the same. Both take a parameter `last` which says whether the command is displaying the buffer as its last action.

If `last` is nonzero, Epsilon will create the window and then return. Epsilon's main command loop will take care of displaying the pop-up window, scrolling through it, and removing it when the user's done examining it. If the user executes a command like find-file while the pop-up window is still on the screen, Epsilon will remove the pop-up and continue with the command.

If `last` is zero, the viewing subroutine will not return until the user has removed the pop-up window (by pressing ⟨Space⟩ or Ctrl-G, for example). The command can then continue with its processing. The user won't be able to execute a prompting command like find-file while the pop-up window is still on the screen.

```
view_linked_buf(int buf, int last, int (*linker)())
int linker(char *link)  /* linker function prototype */
```

Epsilon uses a variation of `view_buf()` to display some online help. The variation adds support for simple hyperlinks. The user can select one of the links in a page of displayed text and follow it to go to another page, or potentially to perform any other action. The `view_linked_buf()` subroutine shows a buffer with links.

The links are delimited with a Ctrl-A character before and a Ctrl-B character after each link. Epsilon's non-Windows documentation file edoc is in this format. (See page 302.) The `view_linked_buf()` subroutine will modify the buffer it receives, removing and highlighting the links before displaying it.

When the user follows a link, Epsilon will call the function pointer `linker` passed as a parameter to `view_linked_buf()`. The `linker` function, which may have any name, will receive the link text as a parameter.

```
    /* space at sides of viewed popup */
short _view_left = 2;
short _view_top = 2;
short _view_right = 2;
short _view_bottom = 6;

short _view_border = ALL_BORD(BSINGLE);
char *_view_title;     /* title for viewed popup */
int view_loop(int win)
```

By default, the above subroutines create a pop-up window with no title and a single-line border, almost filling the screen. The window begins two columns from the left border and stops two columns from the right, and extends two lines from the top of the screen to six lines from the bottom. You can alter any of these values by setting the variables `_view_title`, `_view_border`, `_view_left`, `_view_top`, `_view_right`, and `_view_bottom`. Preserve the original default value using the `save_var` keyword. For example, this code fragment shows a buffer in a narrow window near the right edge of the screen labeled "Results" (surrounding a title with spaces often makes it more attractive):

```
save_var _view_left = 40;
save_var _view_title = " Results ";
save_var _view_border = ALL_BORD(BDOUBLE);
view_buffer(buf, 1);
```

A command that displays a pop-up window may want more control over the creation and destruction of the pop-up window than `view_buf()` and similar subroutines provide. A command can instead create its pop-up window itself, and call `view_loop()` to handle user interaction. The `view_loop()` subroutine takes the handle of the pop-up window to work with. The pop-up window may be a part of a dialog. (See the `display_dialog_box()` primitive described on page 322.)

The `view_loop()` subroutine lets the user scroll around in the window and watches for an unrecognized key (an alphabetic key, for example) or a key that has a special meaning. It returns when the user presses one of these keys or when the user says to exit. By default, the user can scroll off either end of the buffer and this subroutine will return. Set the `paging-retains-view` variable nonzero to prevent this. The `view_loop()` subroutine returns an `INP_` code from eel.h to indicate which user action made it exit. See that file for more information. The function that called `view_loop()` may choose to call `view_loop()` again, or to destroy the pop-up window and continue.

```
error_if_input(int abort) /* complete.e */
remove_final_view() /* complete.e */
```

If the user is entering a response to some prompt and gives another command that also requires a response, Epsilon aborts the command to prevent confusion. Such commands should call `error_if_input()`, which will abort if necessary. The subroutine also removes a viewed buffer, as described above, by calling `remove_final_view()` if necessary. If its `abort` parameter is nonzero, it will attempt to abort the outer command as well, if aborting proves necessary.

8.2.6   Window Attributes

```
int get_wattrib(int win, int code)
set_wattrib(int win, int code, int val)
/* use these codes with get_wattrib() & set_wattrib() */
#define BLEFT            0
#define BTOP             1
#define BRIGHT           2
#define BBOTTOM          3
#define PBORDERS         4
#define PHORIZBORDCOLOR 5
#define PVERTBORDCOLOR  6
#define PTEXTCOLOR       7
#define PTITLECOLOR      8
```

The get_wattrib() and set_wattrib() primitives let you examine and modify many of a window's attributes, such as its position, size, or color. The win parameter contains the handle or number of the window to modify, and the code parameter specifies a particular attribute.

For the code parameter, you can specify one of BLEFT, BTOP, BRIGHT, or BBOTTOM, to examine or change the window's size or position. They refer to the screen coordinate of the corresponding edge. You can use PBORDERS to specify a new border code (see the description of add_popup() above). Or you can set one of the window's colors: each window has a particular color class it uses for its normal text (outside of any highlighted regions), its horizontal borders, its vertical borders, and its title text. Use the macros PTEXTCOLOR, PHORIZBORDCOLOR, PVERTBORDCOLOR, and PTITLECOLOR, respectively, to refer to these. Set them using a color class expression. (See page 90.) For example, the statement

```
set_wattrib(win, PTEXTCOLOR, color_class viewed_text);
```

makes the text in window win appear in the color the user selected for viewed text.

```
window char system_window;
window char invisible_window;
```

Setting the window-specific primitive variable system_window to a nonzero value designates the current window as a system window. The user commands that switch windows will skip over system windows. Setting the window-specific primitive variable invisible_window to a nonzero value makes a window whose text Epsilon won't display (although it will display the border, if the window has one). Epsilon won't modify the part of the screen that would ordinarily display the window's text.

8.2.7 Buffer Text in Windows

```
to_buffer(char *buf)          /* buffer.e */
to_buffer_num(int bnum)     /* buffer.e */
window short window_bufnum;
switch_to_buffer(int bnum)
int give_prev_buf()         /* buffer.e */
to_another_buffer(char *buf)
tiled_only()             /* window.e */
int in_bufed()              /* bufed.e */
quit_bufed()             /* bufed.e */
```

The to_buffer() subroutine defined in buffer.e connects the current window to the named buffer, while to_buffer_num() does the same, but takes a buffer number. Both work by setting the window_bufnum variable, first remembering the previous buffer displayed in the window so the user can easily return to it. The window_bufnum variable stores the buffer number of the buffer displayed in the current window.

Both of these functions check the file date of the new buffer and warn the user if the buffer's file has been modified on disk. The switch_to_buffer() subroutine skips this checking.

The give_prev_buf() subroutine retrieves the saved buffer number of the previous buffer displayed in the current window. If the previous buffer has been deleted, or there is no previous buffer for this window, it returns the number of another recently-used buffer. If it can't find any suitable buffer, it returns 0.

The to_another_buffer() subroutine makes sure that buf is not the current buffer. If it is, the subroutine switches the current window to a different buffer. This subroutine is useful when you're about to delete a buffer.

Sometimes the user may issue a command that switches buffers, while in a bufed pop-up window, or some other type of pop-up window. Issuing `to_buffer()` would switch the pop-up window to the new buffer, rather than the underlying window. Such commands should call the `tiled_only()` subroutine before switching buffers. This subroutine removes any bufed windows or other unwanted windows, and returns to the original tiled window. It calls the `quit_bufed()` subroutine to remove bufed windows. If it can't remove some pop-up windows, it tries to abort the command that created them. The `quit_bufed()` subroutine uses the `in_bufed()` subroutine to determine if the current window is a bufed window.

```
user window int window_start;
user window int window_end;
fix_window_start()       /* window.e */
```

The `window_start` variable provides the buffer position of the first character displayed in the current window. Epsilon's redisplay sets this variable, but you can also set it manually to change what part of the buffer appears in the window. When Epsilon updates the window after a command, it makes sure that point is still somewhere on the screen, using the new value for `window_start`. If not, it alters `window_start` so point is visible.

The `window_end` variable provides the buffer position of the last character displayed in the window. Epsilon's redisplay sets this variable. Setting it does nothing.

The `fix_window_start()` subroutine adjusts `window_start`, if necessary, so that it occurs at the beginning of a line.

```
int get_window_pos(int pos, int *row, int *col)
int window_line_to_position(int row)
```

The `get_window_pos()` function takes a buffer position and finds the window row and column that displays the character at that position. It puts the row and column in the locations that `row` and `col` point to. It returns 0 if it could find the position in the window, or a code saying why it could not.

A return value of 1 means that the position you gave doesn't appear in the window because it precedes the first position displayed in the window. If the given position doesn't appear in the window because it follows the last position displayed in the window, the function returns 2. A return value of 3 means that the position "appears" before the left edge of the screen (due to horizontal scrolling), and 4 means that the position "appears" too far to the right. It doesn't change the locations that `row` and `col` refer to when it returns 1 or 2.

The `window_line_to_position()` primitive takes the number of a row in the current window, and returns the buffer position of the first character displayed on that row. It returns -1 if the row number provided is negative or greater than the number of rows in the window.

```
user int line_in_window;
user int column_in_window;
```

The `line_in_window` and `column_in_window` primitives give you the position of point in the current window, as set by the last `refresh()`. Both variables start counting from 0. If you switch windows, Epsilon will not update these variables until the next `refresh()`.

```
int window_extra_lines()
build_window()
window_to_fit(int max)       /* window.e */
popup_near_window(int new, int old)
```

When buffer text doesn't reach to the bottom of a window, Epsilon blanks the rest of the window. The `window_extra_lines()` primitive gives the number of blank lines at the bottom of the window that don't correspond to any lines in the buffer.

Some of the functions that return information about the text displayed in a window only provide information as of the last redisplay. Due to buffer changes, their information may now be outdated. The `build_window()` primitive reconstructs the current window internally, updating Epsilon's idea of which lines of text go where in the window, how much will fit, and so forth. This primitive updates the value of `window_end`. It may also modify the `display_column` and `window_start` variables if displaying the window as they indicate doesn't get to `point`. The `build_window()` function also updates the values returned by the `window_line_to_position()`, `get_window_pos()`, and `window_extra_lines()` functions.

Use the `window_to_fit()` subroutine to ensure that a pop-up window is no taller than it needs to be. It sets the window's height so that it's just big enough to hold the buffer's text, but never more than `max` lines tall. The subroutine has no effect on windows that form part of a dialog.

The `popup_near_window()` subroutine tries to move a pop-up window on the screen so it's near another window. It also adjusts the height of the pop-up window based on its contents, by calling `window_to_fit()`. The bufed command uses this to position its pop-up buffer list near the tiled window from which you invoked it.

```
window_scroll(int lines)
```

The `window_scroll()` primitive scrolls the text of the current window up or down. It takes an argument saying how many lines up to scroll the current window. With a negative argument, this primitive scrolls the window down. (See page 228 for information on scrolling text left or right.)

8.2.8 Window Titles and Mode Lines

```
window_title(int win, int edge, int pos, char *title)
#define TITLECENTER          (0)
#define TITLELEFT(offset)     (1 + (offset))
#define TITLERIGHT(offset)    (-(1 + (offset)))
make_title(char *result, char *title, int room)
```

You can position a title on the top or bottom border of a window using the `window_title()` primitive. (Also see the `set_window_caption()` primitive described on page 324.) It takes the window number in `win` and the text to display in `title`. (It makes a copy of the text, so you don't need to make sure it stays around after your function returns.) The `edge` parameter must have the value of `BTOP` or `BBOTTOM`, depending on whether you want the title displayed on the top or bottom border of the window.

Construct the `pos` parameter using one of the macros `TITLELEFT()`, `TITLECENTER`, or `TITLERIGHT()`. The `TITLECENTER` macro centers the title in the window. The other two take a number which says how many characters away from the given border the title should appear. For example, `TITLERIGHT(3)` puts the title three characters away from the right-hand edge of the window.

Epsilon interprets the percent character '%' specially when it appears in the title of a window. Follow the percent character with a character from the following list, and Epsilon will substitute the indicated value for that sequence:

%c Epsilon substitutes the current column number, counting columns from 0.

%C Epsilon substitutes the current column number, counting columns from 1.

%d Epsilon substitutes the current display column, with a < before it, and a space after. However, if the display column has a value of 0 (meaning horizontal scrolling is enabled, but the window has not been scrolled), or −1 (meaning the window wraps long lines), Epsilon substitutes nothing.

%D Epsilon substitutes the current display column, but if the display column is −1, Epsilon substitutes nothing.

%l Epsilon substitutes the current line number.

%m Epsilon substitutes the text " More ", but only if characters exist past the end of the window. If the last character in the buffer appears in the window, Epsilon substitutes nothing.

%P Epsilon substitutes the percentage of point through the buffer, followed by a percent sign.

%p Epsilon substitutes the percentage of point through the buffer, followed by a percent sign. However, if the bottom of the buffer appears in the window, Epsilon displays Bot instead. Epsilon displays Top if the top of the buffer appears, and All if the entire buffer is visible.

%s Epsilon substitutes "* " if the buffer's `modified` flag has a nonzero value, otherwise nothing.

%S Epsilon substitutes "*" if the buffer's `modified` flag has a nonzero value, otherwise nothing.

%h Epsilon substitutes the current hour in the range 1 to 12.

%H Epsilon substitutes the current hour in military time in the range 0 to 23.

%n Epsilon substitutes the current minute in the range 0 to 59.

%e Epsilon substitutes the current second in the range 0 to 59.

%a Epsilon substitutes "am" or "pm" as appropriate.

Note: For the current time, use a sequence like `%2h:%02n %a` for "3:45 pm" or `%02H:%02n:%02e` for "15:45:21".

%% Epsilon substitutes a literal "%" character.

%< Indicates that redisplay may omit text to the left, if all of the information will not fit.

%> Puts any following text as far to the right as possible.

   With any of the numeric sequences, you can include a printf-style field width specifier between the `%` and the letter. You can use the same kinds of field width specifiers as C's `printf()` function. In column 9, for example, the sequence `%4c` expands to "   9", `%04c` expands to "0009", and `%-4c` expands to "9   ".

   You can expand title text in the same way as displaying it would, using the `make_title()` primitive. It takes the title to expand, a character array where it will put the resulting text, and a width in which the title must fit. It returns the actual length of the expanded text.

```
prepare_windows()        /* disp.e */
window char _window_flags;
#define FORCE_MODE_LINE 1
#define NO_MODE_LINE    2
#define WANT_MODE_LINE  4

build_mode()             /* disp.e */
assemble_mode_line(char *line) /* disp.e */
```

```
set_mode(char *mode)          /* disp.e */
buffer char *major_mode; /* EEL variable */
user char mode_format[60];
```

Whenever Epsilon thinks a window's mode line or title may be out of date, it arranges to call the `prepare_windows()` and `build_mode()` subroutines during the next redisplay. The `prepare_windows()` subroutine arranges for the correct sort of borders on each window. This sometimes depends on the presence of other windows. For example, tiled windows get a right-hand border only if there's another window to their right. This subroutine will be called before text is displayed.

By default, `prepare_windows()` puts a mode line on all tiled windows, but not on any pop-up windows. You can set flags in the window-specific `_window_flags` variable to change this. Set `FORCE_MODE_LINE` if you want to put a mode line on a pop-up window, or set `NO_MODE_LINE` to suppress a tiled window's mode line. The `prepare_windows()` subroutine interprets these flags, and alters the `WANT_MODE_LINE` flag to tell `build_mode()` whether or not to put a mode line on the window.

The `build_mode()` subroutine calls the `assemble_mode_line()` subroutine to construct a mode line, and then uses the `window_title()` primitive to install it.

The `assemble_mode_line()` subroutine calls the `set_mode()` subroutine to construct the part of the mode line between square brackets (the name of the current major mode and a list of minor modes).

While many changes to the mode line require a knowledge of EEL, you can do some simple customizations by setting the variable `mode_format`. Edit these variables with set-variable, using the percent character sequences listed above. For example, if you wanted each mode line to start with a line and column number, you could add the text " Line %l Col %c " to `mode_format`.

An EEL function can add text to the start of a particular buffer's mode line by setting the buffer-specific variable `mode_extra`. Call the `set_mode_message()` subroutine to do this. It takes a pointer to the new text, or `NULL` to remove the current buffer's extra text. Internet FTP's use this to display the percent of a file that's been received (and similar data).

The `set_mode()` subroutine gets the name of the major mode from the buffer-specific `major_mode` variable, and adds the names of minor modes itself. You can add new minor modes by replacing this function (see page 293).

```
display_more_msg(int win)
```

The `display_more_msg()` subroutine makes the bottom border of the window `win` display a "More" message when there are characters past the end of the window, by defining a window title that uses the `%m` sequence.

8.2.9 Normal Buffer Display

Epsilon provides many primitives for altering the screen contents. This section describes those relating to the automatic display of buffers that happens after each command, as described below.

```
refresh()
maybe_refresh()
```

The `refresh()` primitive does a standard screen refresh, showing the contents of all Epsilon windows. The `maybe_refresh()` primitive calls `refresh()` only if there is no type-ahead. This is usually preferred since it lets Epsilon catch up with the user's typing more quickly. Epsilon calls the latter primitive after each command executes.

```
user window char build_first;
user buffer char must_build_mode;
user char full_redraw;
user char all_must_build_mode;
```

Epsilon normally displays each window line by line, omitting lines that have not changed. When a command has moved point out of the window, Epsilon must reposition the display point (the buffer position at which to start displaying text) to return point to the window. However, Epsilon sometimes does not know that repositioning is required until it has displayed the entire window. When it discovers that point is not in the window, Epsilon moves the display point to a new position and immediately displays the window again. Certain commands which would often cause this annoying behavior set the `build_first` variable to prevent it.

When the `build_first` variable is set, the next redisplay constructs each window internally first, checks that point is in the window, and only then displays it. The variable is then set back to zero. A `build_first` redisplay is slower than a normal redisplay, but it never flashes an incorrect window.

Epsilon "precomputes" most of the text of each mode line, so it doesn't have to figure out what to write each time it updates the screen. Setting the `must_build_mode` variable to 1 warns Epsilon that any mode lines for the current buffer must be rebuilt. The `make_mode()` subroutine in disp.e sets this to 1, and Epsilon rebuilds the mode lines of all windows displaying this buffer.

Setting the `all_must_build_mode` variable to 1 is like setting `must_build_mode` to 1 for all buffers.

Setting the `full_redraw` variable rebuilds all mode lines, as well as any precomputed information Epsilon may have on window borders, screen colors, and so forth.

It is necessary to set `full_redraw` when two parameters affecting the display have been changed. Make the `full_redraw` variable nonzero if the size of the tab character has changed, or if the display class of any character has been changed via the `_display_class` array.

```
screen_messed()
```

The `screen_messed()` primitive causes the next `refresh()` to completely redraw the entire screen.

```
user window int display_column;
```

The window-specific variable `display_column` determines how Epsilon displays long lines. If negative, Epsilon displays buffer lines too big to fit on one screen line on multiple screen lines, with a \ or graphic character (see the `_display_characters` variable described below) to indicate that the line has been wrapped. If `display_column` is 0 or positive, Epsilon only displays the part of a line that fits on the screen. Epsilon also skips over the initial `display-column` columns of each line when displayed. Horizontal scrolling works by adjusting the display column.

```
int next_screen_line(int n)
int prev_screen_line(int n)
```

The `next_screen_line()` primitive assumes point is at the beginning of a screen line, and finds the nth screen line following that one by counting columns. It returns the position of the start of that line.

The `prev_screen_line()` primitive is similar. It returns the start of the nth screen line before the one point would be on. It does not assume that point is at the start of a screen line.

If Epsilon is scrolling long lines of text rather than wrapping them (because `display_column` is greater than or equal to zero), these primitives go to the beginning of the appropriate line in the buffer, not the `display_column`'th column. In this mode, `next_screen_line(1)` is essentially the same as `nl_forward()`, and `prev_screen_line(0)` is like `to_begin_line()`.

Screen Dimensions

```
short screen_cols;
short screen_lines;
```

The `screen_cols` and `screen_lines` primitives contain the number of columns and lines on Epsilon's main display (the OS window's size, or the current terminal's size, as appropriate). They are set when Epsilon starts up, using values provided by the operating system (or, for the Windows version, by the registry). Don't set these variables directly. Use the `resize_screen()` primitive described below.

```
short want_cols;
short want_lines;
```

The `want_cols` and `want_lines` primitives contain the values the user specified through the -vc and -vl switches, respectively, described on page 14. If these variables are 0, it means the user did not explicitly specify the number of lines or columns to display.

```
term_init()           /* video.e */
term_cmd_line()       /* video.e */
term_mode(int active)  /* video.e */
```

Epsilon's standard startup code calls the subroutine `term_init()` when you start Epsilon, and `term_cmd_line()` when it wants to change Epsilon's window size to match what the user specified on the command line. (It changes sizes based on the command line *after* it restores any saved session.) The `term_mode()` subroutine controls switching when you exit Epsilon or run a subprocess. Its argument is 1 when entering Epsilon again (when a `shell()` call returns, for example) and 0 when exiting.

```
resize_screen(int lines, int cols)
```

Use the `resize_screen()` primitive to tell Epsilon to display a different number of lines or columns. It scales all the windows to the new screen dimensions, and then sets the `screen_lines` and `screen_cols` variables to the new screen size.

Character Display

```
buffer char *_display_class;
user buffer short tab_size;
char *_echo_display_class;
```

Modifying the character array `_display_class` lets you alter the way Epsilon displays characters. There is one position in the array for each of the 256 possible characters in a buffer. The code at each position determines how Epsilon displays the character when it appears in a buffer. This code is a *display code*.

Epsilon lets each character occupy one or more screen positions. For example, the Control-A character is usually shown in two characters on the screen as "^A". The number of columns the ⟨Tab⟩ character occupies depends on the column it starts in. Epsilon uses the display codes 0 through 6 to produce the various multi-character representations it is capable of, as described below.

Besides these multi-character display codes, Epsilon provides a way to have one character display as another. If the display code of a character is not one of the special display codes 0 through 6, Epsilon

interprets the display code as a graphics character. This graphics character becomes the single-column representation.

For example, if the display code for 'A' is 'B' (that is, if the value of `_display_class['A']` is the character 'B'), wherever an 'A' appears in the buffer, a 'B' will appear on the screen when it is displayed. The character is still really an 'A', however: only searches for 'A' will find it, an 'A' will be written if you save the file, and so forth. This facility is especially useful for supporting national character sets.

If a display code is from 0 to 6, it has a special meaning. By default, all characters have such a display code. These numbers have been given names in the file codes.h, and we'll use the names in this discussion for clarity.

Epsilon displays a character with display code `BNORMAL` as the character itself. If character 65, the letter 'A', has display code `BNORMAL` it is the same as if it had display code 65.

Epsilon displays a character with display code `BTAB` as a tab. The character is displayed as the number of blanks necessary to reach the next tab stop. The buffer-specific primitive variable `tab-size` sets the number of columns from one tab stop to the next. By default its value is eight.

A character with display code `BNEWLINE` goes to the start of the next line when displayed, as newline does normally.

Epsilon displays a character with display code `BC` as a control character. It is displayed as the ˆ character, followed by the original character exclusive-or'ed with 64, and with the high bit stripped. `BM` and `BMC` are similar, with the prefix being M- and M-ˆ, respectively.

Finally, Epsilon displays a character with display code `BHEX` as a hexadecimal number in the form 'xB7'. Specifically, the representation has the letter 'x', then the two-character hexadecimal character code. You can change many of the characters Epsilon uses for its representations of newlines, tabs, hex characters, and so forth; see below.

By default, the tab character has code `BTAB`, the newline character has code `BNEWLINE`, and the other control characters have code `BC`. Control characters with the eighth bit set have code `BMC`. All other characters have code `BNORMAL`.

The variable `_display_class` is actually a buffer-specific pointer to the array of display codes. Normally, all these pointers refer to the same array, contained in the variable `_std_disp_class` defined in cmdline.e. You can create other arrays if you wish to have different buffers display characters in different ways. Whenever you change the `_display_class` variable, `build_first` must be set to make the change take effect, as described above.

When displaying text in the echo area, Epsilon uses the display class array pointed to by the `_echo_display_class` variable. It can have the same values as `_display_class`.

```
char _display_characters[ ];
buffer char *buffer_display_characters;
```

It is possible to change the characters Epsilon uses to display certain parts of the screen such as the border between windows. Epsilon gets such characters from the `_display_characters` array. This array contains the line-drawing characters that form window borders, the characters Epsilon uses in some of the display modes set by set-show-graphic, the characters it uses to construct the scroll bar, and the characters Epsilon replaces for the graphical mouse cursor it normally uses in DOS. The set-display-characters command may be used to set these characters.

If the buffer-specific variable `buffer_display_characters` is non-null in a buffer, Epsilon uses it in place of the `_display_characters` variable whenever it displays that buffer. You can use this to provide a special window border, scroll bar, or similar for a particular buffer. Epsilon's change-show-spaces command uses this variable, too.

```
int expand_display(char *to, char *from)
```

The `expand_display()` primitive expands characters to the multicharacter representations they would have if displayed on the screen. It returns the length of the result.

Character Widths and Columns

```
int display_width(int ch, int col)
move_to_column(int col)
int column_to_pos(int col)
```

The number of characters that fit on each screen line depends on the display codes of the characters in the line. Epsilon moves characters with multi-character representations as a unit to the next screen line when they don't fit at the end of the previous one (except in horizontal scrolling mode). Tab characters also vary in width depending upon the column they start in. There are several primitives that count screen columns using display class information.

The `display_width()` primitive is the simplest. It returns the width a character `ch` would have if it were at column `col`. The `move_to_column()` primitive moves to column `col` in the current line, or to the end of the line if it does not reach to column `col`. The `column_to_pos()` subroutine accepts a column number but doesn't move point; instead it returns the buffer position of that column.

```
int horizontal(int pos)
int current_column()
int get_column(int pos)         /* indent.e */
int get_indentation(int pos)    /* indent.e */
int give_position_at_column(int p, int col)
to_column(int col)              /* indent.e */
indent_to_column(int col)       /* indent.e */
indent_like_tab()               /* indent.e */
```

The `horizontal()` primitive returns the number of columns from point to position `pos`. Point doesn't change. It must be before `pos`. The primitive returns −1 if there is a character of display code BNEWLINE between point and `pos`. This primitive assumes that point is in column 0.

The `current_column()` primitive uses the `horizontal()` primitive to return the number of the current column.

The `get_column()` subroutine returns the column number of a given buffer position. The `get_indentation()` subroutine returns the indentation of the line containing position `pos`.

The `give_position_at_column()` subroutine returns the buffer position of the character at column `col` on the line containing position `p`; the column may be −1 to retrieve the position of the end of `p`'s line.

The `to_column()` subroutine indents so that the character immediately after point winds up in column `col`. It replaces any spaces and tabs before point with the new indentation. It doesn't modify any characters after point.

The `indent_to_column()` subroutine indents so that the next non-whitespace character on the line winds up in column `col`. It replaces any spaces and tabs before or after point.

The `indent_like_tab()` subroutine indents like inserting a ⟨Tab⟩ character at point would. However, it respects the `indent-with-tabs` variable and avoids using tabs when the variable is zero. It also converts spaces and tabs immediately before point so that they match `indent-with-tabs` and use the minimum number of characters.

```
force_to_column(int col)      /* indent.e */
```

The `force_to_column()` subroutine tries to move to column `col`. If the line doesn't reach to that column, the function indents out to the column. If the column occurs inside a tab character, the function converts the tab to spaces.

```
user window short cursor_to_column;
to_virtual_column(int col)  /* basic.e */
int virtual_column()         /* basic.e */
int virtual_mark_column()   /* basic.e */
```

The window-specific `cursor_to_column` variable lets you position the cursor in a part of a window where there are no characters. It's normally −1, and the cursor stays on the character after point. If it's non-negative in the current window, Epsilon puts the cursor at the specified column in the window instead. Epsilon resets `cursor_to_column` to −1 whenever the buffer changes, or point moves from where it was when you last set `cursor_to_column`. (Epsilon only checks these conditions when it redisplays the window, so you can safely move point temporarily.)

Similarly, the window-specific `mark_to_column` variable lets you position the mark in a part of a window where there are no characters. Epsilon uses this variable when it displays a region that runs to the mark's position, and swaps the variable with `cursor_to_column` when you exchange the point and mark. It's normally −1, so Epsilon highlights up to the actual buffer position of the mark. If it's non-negative in the current window, Epsilon highlights up to the specified column instead. Epsilon resets `mark_to_column` to −1 just as described above for `cursor_to_column`.

The `to_virtual_column()` subroutine positions the cursor to column `col` on the current line. It tries to simply move to the correct position in the buffer, but if no buffer character begins at that column, it uses the `cursor_to_column` variable to get the cursor to the right place.

The `virtual_column()` subroutine provides the column the cursor would appear in: either the value of the `cursor_to_column` variable, or (if it's negative) the current column. Similarly, the `virtual_mark_column()` subroutine provides the column for the mark, taking `mark_to_column` into account.

```
tab_convert(int from, int to, int totabs)
hack_tabs(int offset)
int maybe_indent_rigidly(int rev)
```

The `tab_convert()` subroutine converts tabs to spaces in the specified region when its parameter `totabs` is zero. When `totabs` is nonzero, it converts spaces to tabs.

The `hack_tabs()` subroutine converts tabs to spaces in the `offset` columns following point. If `offset` is negative, the function converts tabs in the columns preceding point.

Commands bound to ⟨Tab⟩ often call the `maybe_indent_rigidly()` subroutine. If a region's been highlighted, this subroutine indents it using the indent-rigidly command and then returns nonzero. Otherwise, it returns zero. If its parameter `rev` is nonzero, the subroutine unindents; a command bound to Shift-⟨Tab⟩ often provides a nonzero `rev`, but for commands on ⟨Tab⟩ this is typically zero.

```
buffer int (*indenter)(); /* EEL variable */
user buffer int auto_indent;   /* EEL variable */
prev_indenter()                /* indent.e */
```

The `normal-character` command provides a hook for automatic line indentation when it inserts the newline character. If the buffer-specific variable `auto-indent` is nonzero, the `normal-character` command will call the function pointed to by the variable `indenter`, a buffer-specific function pointer, after inserting a newline character. By default, it calls the `prev_indenter()` subroutine, which indents to the same indentation as the previous line.

8.2.10 Displaying Status Messages

```
int say(char *format, ...)
int sayput(char *format, ...)
```

The `say()` primitive displays text in the echo area. It takes a printf-style format string, and zero or more other parameters, as described on page 235. The `sayput()` primitive is similar, but it positions the cursor at the end of the string. Each returns the number of characters displayed.

```
int note(char *format, ...)
int noteput(char *format, ...)
int unseen_msgs()
drop_pending_says()
short expire_message;
```

When you use the `say()`, `sayput()`, or `error()` primitives (`error()`'s description appears on page 284) to display a message to the user, Epsilon ensures that it remains on the screen long enough for the user to see it (the `see-delay` variable controls just how long) by delaying future messages. Messages that must remain on the screen for a certain length of time are called *timed messages*.

The `note()` and `noteput()` primitives work like `say()` and `sayput()`, respectively, but their messages can be overwritten immediately. These untimed messages should be used for "status" messages that don't need to last ("95% done", for example).

Epsilon copies the text of each timed message to the `#messages#` buffer. It doesn't copy untimed messages (but see the `show_text()` primitive below).

The `unseen_msgs()` primitive returns the number of unexpired timed messages. When the user presses a key, and there are unseen messages, Epsilon immediately displays the most recent message waiting to be displayed, and discards all pending timed messages.

The `drop_pending_says()` primitive makes Epsilon discard any timed messages that have not yet been displayed. It also makes the current message be untimed (as if it were generated by `note()`, not `say()`), so that the next `say()`, `note()`, or similar will appear immediately. It returns 0 if there were no timed messages, or 1 if there were (or the current message had not yet expired).

An EEL function sometimes needs to display some text in the echo area that is only valid until the user performs some action. For instance, a command that displays the number of characters in the buffer might wish to clear that count if the user inserts or deletes some characters. After displaying text with one of the primitives above, an EEL function may set the `expire_message` variable to 1 to tell Epsilon to clear that text on the next user key.

```
int show_text(int column, int time, char *fmt, ...)
```

The `show_text()` primitive is the most general command for displaying text in the echo area. Like the other display primitives, it takes a printf-style format string, and returns the number of characters it displayed.

When Epsilon displays text in the echo area, you can tell it to begin at a particular column, and Epsilon will subdivide the echo area into two sections. You can then display different messages in each area independently of one another. When it's necessary to display a very long message, Epsilon will combine the sections again and use the full display width. There are never more than two sections in the echo area.

In detail, the `show_text()` primitive tells Epsilon to begin displaying text in the echo area at the specified column, where the leftmost column is column 0. Epsilon then clears the rest of that echo area section, but doesn't modify the other section.

Whenever you specify a column greater than zero in `show_text()`, Epsilon will subdivide the echo area at that column. It will clear any text to the right of the newly-displayed text, but not any text to its left.

Epsilon will recombine the sections of the echo area under two conditions: whenever you write text starting in column 0 that begins to overwrite the next section, and whenever you write the empty string `""` at column 0. When Epsilon recombines sections, it erases the entire echo area before writing the new text.

Specifying a column of −1 acts just like specifying column 0, making Epsilon display the text at the left margin, but it also tells Epsilon to position the cursor right after the text.

The `time` says how long in hundredths of a second Epsilon must display the message before moving on and displaying the next message, if any. As with any timed message, when the user presses a key, Epsilon immediately displays the last message waiting, skipping through any pending messages. A value of 0 for `time` means the message doesn't have to remain for any fixed length of time. A value of −1 means that Epsilon may not go on to the next message until it receives a keystroke; such messages will never time out.

Most of the other echo area display primitives are equivalent to some form of `show_text()`, as shown in the following table:

```
note("abc")            show_text(0, 0, "abc")
say("abc")             show_text(0, see_delay, "abc")
noteput("abc")         show_text(-1, 0, "abc")
sayput("abc")          show_text(-1, see_delay, "abc")
```

Just as Epsilon copies timed messages created with `say()` or `sayput()` to the `#messages#` buffer, the text from a `show_text()` call will be copied if its time is nonzero. Epsilon treats a `time` of 1 (hundredth of a second) the same as zero (it's untimed), but still copies it to the `#messages#` buffer. A column of −2 has a special meaning; Epsilon copies the resulting text to the `#messages#` buffer if `time` is nonzero, but doesn't display it at all.

```
delayed_say(int flags, int before, int after, char *fmt, ...)
```

The `delayed_say()` primitive tells Epsilon to display some text later. It's intended for operations that may take some time. EEL code may display a message provisionally before starting some potentially lengthy operation, telling Epsilon to display it only after a certain amount of time has elapsed, and then cancel the pending message when it finishes. The message will only appear if the operation actually took a long time.

The `before` parameter says how long to wait, in hundredths of a second, before displaying the message, which is built using the printf-style format string `fmt` and any following parameters. If the resulting message is zero-length, it simply cancels any delayed say message that has not yet been displayed.

If the `before` time elapses and the message hasn't been canceled, Epsilon displays it, ensuring it isn't overwritten by a following message for at least `after` hundredths of a second. Epsilon saves the message in the `#messages#` buffer if `after` is nonzero. If `flags` is 1, Epsilon positions the cursor after the message; otherwise it doesn't.

```
int mention(char *format, ...)
user char mention_delay;
```

The `mention()` primitive acts like `sayput()`, but displays its string only after Epsilon has paused waiting for user input for `mention_delay` tenths of a second. It doesn't cause Epsilon to wait for input, it just arranges things so that if Epsilon does wait for input and the required delay elapses, the message is displayed and the wait continues. Writing to the echo area with `say()` or the like cancels any pending `mention()`. By default, `mention_delay` is 0.

```
int muldiv(int a, int b, int c)
```

The `muldiv()` primitive takes its arguments and returns the value $a * b/c$, performing this computation using 64-bit arithmetic. It's useful in such tasks as showing "percentage complete" while operating on a large buffer. Simply writing `point * 100 / size()` in EEL would use 32-bit arithmetic, as EEL always does, and on large buffers (over about 20 megabytes) the result would be wrong.

8.2.11 Printf-style Format Strings

Primitives like `say()` along with several others take a particular pattern of arguments. The first argument is required. It is a character pointer called the *format string*. The contents of the format string determine what other arguments are necessary.

Characters in the format string are copied to the echo area except where a percent character '%' appears. The percent begins a sequence which interpolates the value of an additional argument into the text that will appear in the echo area. The sequence has the following pattern, in which square brackets [ ] enclose optional items:

```
% [ - ] [ number ] [ . number ] character
```

In this pattern *number* may be either a string of digits or the character '`*`'. If the latter, the next argument provided to the primitive must be an int, and its value is used in place of the digits.

The meaning of the sequence depends on the final character:

c The next argument must be an int. (As explained previously, a character argument is changed to an int when a function is called, so it's fine here too.) The character with that ASCII code is inserted in the displayed text. For example, if the argument is 65 or `'A'`, the letter A appears, since the code for A is 65.

d The next argument must be an int. A sequence of characters for the decimal representation of that number is inserted in the displayed text. For example, if the argument is 65 the characters '6' and '5' are produced.

x The next argument must be an int. A sequence of characters for the hexadecimal (base 16) representation of that number is inserted in the displayed text. For example, if the argument is 65 the characters '4' and '1' are produced (since the hexadecimal number 0x41 is equal to 65 in base 10). No minus sign appears with this representation.

o The next argument must be an int. A sequence of characters for the octal representation of that number is inserted in the displayed text. For example, if the argument is 65 the three characters "101" are produced (since the octal number 101 is equal to 65 in base 10). No minus sign appears with this representation.

s The next argument, which must be a string, is copied to the displayed text.

q The next argument, which must be a string, is copied to the displayed text, but quoted for inclusion in a regular expression. In other words, any characters from the original string that have a special meaning in regular expressions are copied with a percent character ('%') before them. See page 55 for information on regular expressions.

r The next argument (which must be a string containing a file name in absolute form) is copied to the displayed text, after being converted to relative form. Epsilon calls the `relative()` primitive, described on page 262, to do this.

f The next argument must be a string, typically the name of a file. This sequence is just like %s except when used in a primitive that displays text in the echo area, such as `say()`, and the entire text to be displayed is too wide to fit in the available room. In that case, Epsilon calls the `abbreviate_file_name()` subroutine defined in disp.e to abbreviate the file name so the entire message fits in the available width. If the displayed message is also recorded in the `#messages#` buffer, where no width restriction applies, the unabbreviated form of the message will be used.

k The next argument must be a number. Epsilon interprets it as a key code or Unicode character code, and interpolates the name of that key or character. If a number starting with zero appears after the %character, Epsilon uses the short form of the key name, if any. See the `key_value()` primitive to convert in the opposite direction, converting text (in the short form only) back to a key's numeric code.

p The next argument must be a number. Epsilon interprets it as a color class, and any following text appears in that color. This only works on those primitives that insert text into a buffer; the numeric argument is ignored in a `sprintf()` or `say()` or similar.

   The first number, if present, is the width of the field the argument will be printed in. At least that many characters will be produced, and more if the argument will not fit in the given width. If no number is present, exactly as many characters as are required will be used.

   The extra space will normally be put before the characters generated from the argument. If a minus sign is present before the first number, however, the space will be put at the end instead.

   If the first number begins with the digit 0, the extra space will be filled with zeros instead of spaces. A minus sign before the first number is ignored in this case.

   The second number, if present, is the maximum number of characters from the string that will be displayed. For example, each of these lines displays the text, "Just an example":

```
say("Just %.2s example", "another");

say("Just %.*s example", 7-5, "another");
```

   It may be tempting to substitute a string variable for the first parameter of `say()`. For example, when writing a function that displays its argument `msg` and pauses, it may seem natural to write `say(msg);`. This will work fine unless `msg` contains a '%' character. In that case, you will probably get an error message. Use `say("%s", msg);` instead.

```
user char in_echo_area;
```

   The `in_echo_area` variable controls whether the cursor is positioned at point in the buffer, or in the echo area at the bottom of the screen. The `sayput()` primitive sets this variable, `say()` resets it, and it is reset after each command.

8.2.12   Other Display Primitives

```
term_write(int col, int row, char *str, int count,
           int colorclass, int clear)
term_write_attr(int col, int row, int chartowrite,
                int attrtowrite)
term_clear()
term_position(int col, int row)
```

The following primitives provide low-level screen control. The `term_clear()` primitive clears the screen. The `term_position()` primitive positions the cursor to the indicated row and column. The `term_write()` primitive puts characters directly on the screen. It puts `count` characters from `str` on the screen at the `row` and `column` in the specified `colorclass`. If `clear` is nonzero, it clears the rest of the line. The `term_write_attr()` primitive writes a single character at the specified location on the screen. Unlike `term_write()`, which takes a color class, this primitive takes a raw foreground/background color attribute pair. This primitive does nothing in Epsilon for Windows or under the X windowing system. For all these primitives, `row` and `col` start at 0, and the coordinate 0,0 refers to the upper left corner of the screen. If a keyboard macro is running, the `term_` primitives are ignored.

```
fix_cursor()    /* EEL subr. */
user int normal_cursor;
user int overwrite_cursor;
user int virtual_insert_cursor;
user int virtual_overwrite_cursor;
#define CURSOR_SHAPE(top, bot)       ((top) * 1000 + (bot))
#define GUI_CURSOR_SHAPE(height, width, offset) \
           ((offset * 1000 + (height)) * 1000 + (width))
int cursor_shape;
```

During screen refresh, Epsilon calls the EEL subroutine `fix_cursor()` to set the shape of the cursor. The subroutine chooses one of four variables depending upon the current modes, and copies its value into the `cursor_shape` variable, which holds the current cursor shape code. The Windows and X versions set the `gui_cursor_shape` variable in a similar way, from a different set of four variables. All these variables use values constructed by the `GUI_CURSOR_SHAPE()` or `CURSOR_SHAPE()` macros. See page 88 for details on these variables.

```
windows_set_font(char *title, int fnt_code)
```

The `windows_set_font()` primitive displays a Windows font selection dialog, allowing the user to pick a different font. It takes two parameters. `Title` specifies the title of the dialog box to display. The `fnt_code` says whether to set Epsilon's main font (`FNT_SCREEN`), the font for printing (`FNT_PRINTER`), or the font for Epsilon's dialogs (`FNT_DIALOG`). It's only available in the Windows GUI version of Epsilon. See the `has_feature` variable. The Unix version of Epsilon runs a separate program for its font dialog.

```
int using_oem_font(int screen)
char using_new_font;
```

The `using_oem_font()` primitive returns a nonzero value if the specified screen's font uses the OEM character set, rather than the ANSI/Windows character set. It takes a screen number. This primitive always returns 0 under Unix. The primitive variable `using_new_font` will be nonzero whenever some screen's font has been changed since the end of the last screen refresh, or when a new screen has been created, for example by displaying a dialog.

8.2.13 Highlighted Regions

Epsilon can display portions of a buffer in a different color than the rest of the buffer. We call each such portion a region. The most familiar region is the one between point and mark. Epsilon defines this region automatically each time you create a new buffer. (Also see the description of character coloring on page 241.)

Epsilon can display a region in several ways. The most common method corresponds to the one you see when you set the mark (by typing Ctrl-@) and then move around: Epsilon highlights each of the characters between point and mark. If you use the mark-rectangle command on Ctrl-X # to define a rectangular region, the highlighting appears on all columns between point and mark, on all lines between point and mark. The pop-up windows of the completion facility illustrate a third type of highlighting, where complete lines appear highlighted. The header file codes.h defines these types of regions as (respectively) REGNORM, REGRECT, and REGLINE. Epsilon won't do any highlighting for a region that has type 0.

A fourth type of highlighting, REGINCL, is similar to REGNORM, but includes an additional character at the end of the region. If a REGNORM region runs between position 10 and position 20 in the buffer, Epsilon would highlight the 10 characters between the two positions. But if the region were a REGINCL region, it would include 11 characters: the characters at positions 10 and 20, and all the characters between.

```
int add_region(spot from, spot to, int color,
               int type, ?int handle)
remove_region(int handle)
int modify_region(int handle, int code, int val)
window char _highlight_control;
```

You can define new regions with add_region(). It takes a pair of spots, a color class expression such as color_class highlight, a region display type (as described above), and, optionally, a numeric "handle". It returns a nonzero numeric handle which you can use to refer to the region later. You can provide the spots in either order, and you may give the same spot twice (for example, in conjunction with REGLINE, to always highlight a single line). See page 90 for basic information on color classes, and page 176 for details on the syntax of color class expressions).

When you omit the handle parameter to add_region() (or provide a handle of zero) add_region() assigns an unused handle to the new region. You can also provide the handle of an existing region, and add_region() will assign the same handle to the new region. Any changes you make to one region by using modify_region() will now apply to both, and a single remove_region() call will remove both. You can link any number of regions in the same buffer in this way. The special handle value 1 refers to the region between point and mark that Epsilon creates automatically.

The remove_region() primitive takes a region handle, and deletes all regions with that handle. The handle may belong to a region in another buffer. Epsilon signals an error if the handle doesn't refer to any region.

The modify_region() primitive retrieves or sets some of the attributes of one or more regions. It takes a region handle, a modify code (one of the MR... codes below), and a new value. If you provide a "new value" that's out of range (such as −2, out of range for all modify codes), Epsilon will not change that attribute of the region, but will simply return its value. If you provide a valid new value, Epsilon will set that attribute of the region, and will return its previous value.

When several regions share the same handle, it's possible they will have different attribute settings. In this case, which region's attribute Epsilon returns is undefined. If you specify a new value for an attribute, it will apply to all regions with that handle.

The modify code MRCOLOR may be used to get or change a region's color class. The modify code MRTYPE may be used to get or change a region's display type, such as REGRECT. The codes MRSTART and

MREND may be used to set the two spots of a region; however, Epsilon will not return the spot identifier for a region, but rather its current buffer position.

You can force a region's starting and ending positions to specific columns using the modify codes MRSTARTCOL and MRENDCOL. For example, if a region runs from point to mark, and you set its MRSTARTCOL to 3, the region will start at column 3 of whatever line point is on. A column setting of −1 here makes Epsilon use the actual column value of the spot; no column will be forced.

You can set up a region to be "controlled" by any numeric global variable. Epsilon will display the region only if the variable is nonzero. This is especially useful because the variable may be window-specific. Since regions are associated with buffers, this is needed so that a buffer displayed in two windows can have a region that appears in only one of them.

The standard region between point and mark is controlled by the window-specific character variable _highlight_control. By default, other regions are not controlled by any variable. The modify code MRCONTROL may be used with modify_region() to associate a controlling variable with a region. Provide the global variable's name table index (obtainable through find_index()) as the value to set.

```
set_region_type()       /* disp.e */
int region_type()       /* disp.e */
highlight_on()          /* disp.e */
highlight_off()         /* disp.e */
int is_highlight_on()   /* disp.e */
```

Several subroutines let you conveniently control highlighting of the standard region between point and mark. To set the type of the region, call the subroutine set_region_type() with the region type code, one of REGNORM, REGRECT, REGLINE, or REGINCL. This doesn't automatically turn on highlighting. Call highlight_on() to turn on highlighting, or highlight_off() to turn it off.

The region_type() subroutine returns the type of the current region, whether or not it's currently highlighted. The is_highlight_on() subroutine returns the type of the current region, but only if it's highlighted. It returns 0 if highlighting is off.

There are several subroutines that help you write functions that work with different types of regions. If you've written a function that operates on the text of a normal Epsilon region, add the following lines at the beginning of your function to make it work with inclusive regions and line regions as well:

```
save_spot point, mark;
fix_region();
```

When the user has highlighted an inclusive or line region, the fix_region() subroutine will reposition point and mark to form a normal Epsilon region with the same characters. (For example, in the case of a line region, Epsilon moves point to the beginning of the line.) The function also swaps point and mark so that point comes first (or equals mark, if the region happens to be empty). This is often convenient.

This procedure assumes your function doesn't plan to modify point or mark, just the characters between them, and it makes sure that point and mark remain in the same place. If your function needs to reposition the point or mark, try omitting the save_spot line. Your function will be responsible for determining where the point and mark wind up.

A function needs to do more work to operate on rectangular regions. If it's built to operate on all the characters in a region, without regard to rectangles or columns, the simplest approach may be to extract the rectangle into a temporary buffer, modify it there, and then replace the rectangle in the original buffer. Several Epsilon subroutines help you do this. For a concrete example, let's look at the function fill_rectangle(), defined in format.e. The fill-region command calls this function when the current region is rectangular.

```
// Fill paragraphs in rectangle between point and mark
// to marg columns (relative to rectangle's width if <=0).
fill_rectangle(marg)
{
        int width, orig = bufnum, b = tmp_buf();

        width = extract_rectangle(b, 0);
        save_var bufnum = b;
        mark = 0;
        margin_right = marg + (marg <= 0 ? width : 0);
        do_fill_region();
        xfer_rectangle(orig, width, 1);
        buf_delete(b);
}
```

The function begins by allocating a temporary buffer using `tmp_buf()`. Then it calls the `extract_rectangle()` subroutine to copy the rectangle into the temporary buffer. This function returns the width of the rectangle it copied. The call from `fill_rectangle()` passes the destination buffer number as the first parameter. Then `fill_rectangle()` switches to the temporary buffer and reformats the text. Finally, the subroutine copies the text back into its rectangle by calling `xfer_rectangle()` and deletes the temporary buffer. If the operation you want to perform on the text in the rectangle depends on any buffer-specific variables, be sure to copy them to the temporary buffer.

Now let's look at the two rectangle-manipulating subroutines `fill_rectangle()` calls in more detail.

```
extract_rectangle(int copybuf, int remove)
```

The `extract_rectangle()` subroutine operates on the region between point and mark in the current buffer. It treats the region as a rectangle, whether or not `region_type()` returns REGRECT. It can perform several different actions, depending upon its parameters. If `copybuf` is nonzero, the subroutine inserts a copy of the rectangle into the buffer with that buffer number. The buffer must already exist.

If `remove` is 1, the subroutine deletes the characters inside the rectangle. If `remove` is 2, the subroutine replaces the characters with spaces. If `remove` is 0, the subroutine doesn't change the original rectangle.

The subroutine always leaves point at the upper left corner of the rectangle and mark at the lower right. It return the width of the rectangle.

```
xfer_rectangle(int dest, int width, int overwrite)
```

The `xfer_rectangle()` subroutine inserts the current buffer as a rectangle of the given `width` into buffer number `dest`, starting at `dest`'s current point. If `overwrite` is nonzero, the subroutine copies on top of any existing columns. Otherwise it inserts new columns. In the destination buffer, it leaves point at the top left corner of the new rectangle, and mark at the bottom right. The point remains at the same position in the original buffer.

```
rectangle_standardize()
```

Functions that manipulate rectangles can sometimes use the `rectangle_standardize()` subroutine to simplify their logic. In a rectangular region, point may be at any one of the four corners of the rectangle. This subroutine moves point and mark so they indicate the same region, but with point at the lower right and mark at the upper left. It's like the rectangular region equivalent of the `fix_region()` subroutine.

```
do_shift_selects()
```

Commands bound to cursor keys typically select text when you hold down the shift key. They do this by calling `do_shift_selects()` as they start. This routine looks at the current state of the shift key and whether or not highlighting is already on, and turns highlighting on or off as needed, possibly setting point.

```
make_line_highlight()      /* complete.e */
remove_line_highlight()    /* complete.e */
```

The `make_line_highlight()` subroutine uses the `add_region()` primitive to create a region that highlights the current line of the current buffer. When Epsilon puts up a menu of options, it uses this function to keep the current line highlighted. The `remove_line_highlight()` subroutine gets rid of such highlighting.

8.2.14 Character Coloring

You can set the color of individual characters using the `set_character_color()` primitive. At first glance, this feature may seem similar to Epsilon's mechanism for defining highlighted regions. Both let you specify a range of characters and a color to display them with. But each has its own advantages.

Region highlighting can highlight the text in different ways: as a rectangle, expanded to entire lines, and so forth, while character coloring has no similar options. You can define a highlighted region that moves around with the point, the mark, or any other spot. Character coloring always remains with the characters.

But when there are many colored regions, using character coloring is much faster than creating a corresponding set of highlighted regions. If you define more than a few dozen highlighted regions, Epsilon's screen refreshes will begin to slow down. Character coloring, on the other hand, is designed to be very fast, even when there are thousands of colored areas. Character coloring is also easier to use for many tasks, since it doesn't require the programmer to allocate spots to delimit the ends of the colored region, or delete them when the region is no longer needed.

One more difference is the way you remove the coloring. For highlighted regions, you can turn off the coloring temporarily by calling `modify_region()`, or eliminate the region entirely by calling `remove_region()`. To do either of these, you must supply the region's handle, a value returned when the region was first created. On the other hand, to remove character coloring, you can simply set the desired range of characters to the special color -1. A program using character coloring doesn't need to store a series of handles to remove or modify the coloring.

Epsilon's code coloring functions are built on top of the character coloring primitives described in this section. See the next section for information on the higher-level functions that make code coloring work.

```
set_character_color(int pos1, int pos2, int color)
```

The `set_character_color()` primitive makes Epsilon display characters between `pos1` and `pos2` using the specified color class. Epsilon discards any previous color settings of characters in that range.

A color class of -1 means the text will be "uncolored". To display uncolored text, Epsilon uses the standard color class `text`. When a buffer is first created, every character is uncolored.

When you insert text in a buffer, it takes on the color of the character immediately after it, or in the case of the last character in the buffer, the character immediately before it. Characters inserted in an empty buffer are initially uncolored. Copying text from one buffer to another does not automatically transfer the color; Epsilon treats the new characters the same as any other inserted text. You can use the `buf_xfer_colors()` subroutine to copy text from one buffer to another and retain its coloring. See page 199.

Epsilon maintains the character colors set by this primitive independently of the highlighted regions created by `add_region()`. The `modify_region()` primitive will never change what `get_character_color()` returns, and similarly the `set_character_color()` primitive never changes the attributes of a region you create with `add_region()`. When Epsilon displays text, it combines information from both sources to determine the final color of each character.

When displaying a buffer, Epsilon uses the following procedure when determining which color class to use for a character:

- Make a list of all old-style highlighted regions that contain the character, and the color classes used for each.

- Add the character's color as set by `set_character_color()` to this list.

- Remove color classes of −1 from the list.

Next, Epsilon chooses a color class from the list:

- If the list of color classes is empty, use the `text` color class.

- Otherwise, if the list contains the `highlight` color class, use that.

- Otherwise, use the color class from the old-style highlighted region with the highest region number. If there are no old-style highlighted regions in the list, the list must contain only one color class, so use that.

- Finally, if we wound up selecting the `text` color class, and the `text_color` variable isn't equal to `color_class text`, use the color class in the `text_color` variable instead of the `color_class text`.

Notice that when a region using the `highlight` color class overlaps another region, the `highlight` color class takes precedence.

```
short get_character_color(int pos, ?int *startp, ?int *endp)
```

The `get_character_color()` primitive returns the color class for the character at the specified buffer position, as set by `set_character_color()`, or −1 if the character is uncolored, and will be displayed using the window's default color class.

You can also use the primitive to determine the extent of a range of characters all in the same color. If the optional pointer parameters `startp` and `endp` are non-null, Epsilon fills in the locations they point to with buffer positions. These specify the largest region of the buffer containing characters the same color as the one at `pos`, and including `pos`. For example, if the buffer contains a five-character word that has been colored blue, the buffer is otherwise uncolored, and `pos` refers to the second character in the word, then Epsilon will set `*startp` to `pos - 1` and `*endp` to `pos + 4`.

```
set_tagged_region(char *tag, int from, int to, short val)
short get_tagged_region(char *tag, int pos, ?int *from, int *to)
```

The character coloring primitives above are actually built from a more general facility that allows you to associate a set of attributes with a buffer range.

Each set of attributes consists of a tag (a unique string like `"my-tag"`) and, for each character in the buffer, a number that represents the attribute. Each buffer has its own set of tags, and each tag has its own

list of attributes, one for each character. (Epsilon stores the numbers in a way that's efficient when many adjacent characters have the same number, but nothing prevents each character from having a different attribute.)

The `set_tagged_region()` primitive sets the attribute of the characters in the range `from` to `to`, for the specified tag.

The `get_tagged_region()` primitive gets the attribute of the character at position `pos` in the buffer. If you provide pointers `from` and `to`, Epsilon will fill these in to indicate the largest range of characters adjacent to `pos` that have the same attribute as `pos`. Characters whose attributes have never been set for a given tag will have the attribute −1.

Epsilon's character color primitives `set_character_color()` and `get_character_color()` use a built-in tagged region with a tag name of `"colors"`.

8.2.15   Code Coloring Internals

Epsilon's code coloring routines use the character coloring primitives above to do code coloring for various languages like C, TeX, and HTML. There are some general purpose code coloring functions that manage code coloring and decide what sections of a buffer need to be colored. Then, for each language, there are functions that know how to color text in that language.

The general purpose section maintains information on what parts of each buffer have already been colored. It divides each buffer into sections that are already correctly colored, and sections that may not be correctly colored. When the buffer changes, it moves its divisions so that the modified text is no longer marked "correctly colored". Whenever Epsilon displays part of a buffer, this part of code coloring recolors sections of the buffer as needed, and marks them so they won't be colored again unless the buffer changes. Epsilon only displays the buffer after the appropriate section has been correctly colored. This part also arranges to color additional sections of the buffer whenever Epsilon is idle, until the buffer has been completely colored.

The other part of code coloring does the actual coloring of C, TeX, and HTML buffers. You can write new EEL functions to tell Epsilon how to color other languages, and use the code coloring package's mechanisms for remembering which parts of the buffer have already been colored, and which need to be recolored. This section describes how to do this. (Also see page 331.)

```
buffer int (*recolor_range)();
    // how to color part of this buffer
buffer int (*recolor_from_here)();
    // how to find a good starting pos
int color_c_range(int from, int to)
    // how to color part of C buffer
int color_c_from_here(int safe)
    // how to find starting pos in C buffer
buffer char coloring_flags;
#define COLOR_DO_COLORING         1
#define COLOR_IN_PROGRESS         2
#define COLOR_MINIMAL             4
#define COLOR_INVALIDATE_FORWARD  8
#define COLOR_INVALIDATE_BACKWARD 16
#define COLOR_INVALIDATE_RESETS   32
#define COLOR_RETAIN_NARROWING    64
int must_color_through;
```

You must first write two functions and make the buffer-specific function pointers refer to them, in each buffer you want to color. For C/C++/EEL buffers, the c-mode command takes care of setting the function pointers. It also contains the lines

```
if (want_code_coloring)
        when_setting_want_code_coloring();
```

to actually turn on code coloring for the buffer if necessary.

The first function, which must be stored in the buffer-specific `recolor_range` variable, does the actual coloring of a part of the buffer. It takes two parameters `from` and `to` specifying the range of the buffer that needs coloring. It colors at least the specified range, but it may go past `to` and color more of the buffer. It returns the buffer position it reached, indicating that all characters between `from` and its return value are now correctly colored. In C buffers, the `recolor_range` function is named `color_c_range()`.

The `recolor_range` function may decide to mark some characters in the range "uncolored", by calling `set_character_color()` with a color class of `-1`. Or it may assign particular color classes to all parts of the range to be colored. But either way, it should make sure all characters in the given range are correctly colored. Typically, a function begins by setting all characters between `from` and `to` to a default color class, then searching for elements which should be colored differently. Be sure that if you extend the range past `to`, you color all the characters between `to` and your new stopping point.

Epsilon remembers which parts of the buffer require coloring by using a tagged region (see page 242) named "`needs-color`". A coloring routine may decide, while parsing a buffer, that some later or earlier section of the buffer requires coloring; if so, it can set the `needs-color` attribute of that section to `-1` to indicate this, and Epsilon will recolor that section of the buffer the next time it's needed. Or it can declare that some other section of the buffer is already properly colored by setting that section's attribute to `0`. It may also decide to examine the `must_color_through` variable, a buffer position marking the end of the region that really requires coloring right now. (Ordinarily, Epsilon expands this region to include entire color blocks.)

When the buffer's modified, some of its coloring becomes invalid, and must be recomputed the next time it's needed. Normally Epsilon invalidates a few lines surrounding the changed section. Some language modes tell Epsilon to automatically invalidate more of the buffer by setting flags in the buffer-specific `coloring_flags` variable. (Other flags in this variable aren't normally set by language modes; code coloring uses them for bookkeeping purposes.)

`COLOR_INVALIDATE_FORWARD` indicates that after the user modifies a buffer, any syntax highlighting information after the modified region should be invalidated. `COLOR_INVALIDATE_BACKWARD` indicates that syntax highlighting information before the modified region should be invalidated.

`COLOR_INVALIDATE_RESETS` tells Epsilon that whenever it invalidates syntax highlighting in a region, it should also set the color of all text in that region to the default of `-1`. `COLOR_RETAIN_NARROWING` indicates that coloring should respect any narrowing in effect (instead of looking outside the narrowed area to parse the buffer in its entirety).

For many languages, starting to color at an arbitrary place in the buffer requires a lot of unnecessary work. For example, the C language has comments that can span many lines. A coloring function must know whether it's inside a comment before it can begin coloring. Similarly, a coloring function that began looking from the third character in the C identifier `id37` might decide that it had seen a numeric constant, and incorrectly color the buffer.

To simplify this problem, the coloring routines ensure that coloring begins at a safe place. We call a buffer position *safe* if the code coloring function can color the buffer beginning at that point, without looking at any earlier characters in the buffer.

When Epsilon calls the function in `recolor_range`, the value of `from` is always safe. Epsilon expects the function's return value to be safe as well; it must be OK to continue coloring from that point. For C, this means the returned value must not lie inside a comment, a keyword, or any other lexical unit. Moreover, inside the colored region, any boundary between characters set to different color classes must be safe. If the colored region contains a keyword, for example, Epsilon assumes it can begin recoloring from the start of that keyword. (If this isn't true for a particular language, its coloring function can examine the buffer itself to determine where to begin coloring.)

When Epsilon needs to color more of the buffer, it generally starts from a known safe place: either a value returned by the buffer's `recolor_range` function, or a boundary between characters of different colors. But when Epsilon first begins working on a part of the buffer that hasn't been colored before, it must determine a safe starting point. The second function you must provide, stored in the `recolor_from_here` buffer-specific function pointer, picks a new starting point. In C buffers, the `recolor_from_here` function is named `color_c_from_here()`.

The buffer's `recolor_from_here` function looks backward from point for a safe position and returns it. This may involve a search back to the start of the buffer. If Epsilon knows of a safe position before point in the buffer, it passes this as the parameter `safe`. (If not, Epsilon passes 0, which is always safe.) The function should respect the value of the `color-look-back` variable to limit searching on slow machines.

Epsilon provides two standard `recolor_from_here` functions that coloring extensions can use. The `recolor_by_lines()` subroutine is good for buffers where coloring is line-based, such as dired buffers. In such buffers the coloring needed for a line doesn't depend at all on the contents of previous lines. The `recolor_from_top()` subroutine has just the opposite effect; it forces Epsilon to start from the beginning of the buffer (or an already-colored place). This may be all that's needed if a mode's coloring function is very simple and quick.

Epsilon runs the code coloring functions while it's refreshing the screen, so running the EEL debugger on code coloring functions is difficult, since the debugger itself needs to refresh the screen. The best way to debug such functions is to test them out by calling them explicitly, using test-bed functions like these:

```
command debug_color_region()
{
        fix_region();
        set_character_color(point, mark, color_class default);
        point = color_algol_range(point, mark);
}


command debug_from_here()
{
        point = color_algol_from_here(point);
}
```

The first command above tries to recolor the current region, and moves past the region it actually colored. It begins by marking the region with a distinctive color (using the default color class), to help catch missing coloring. The second command helps you test your `from_here` function. It moves point backwards to the nearest safe position. Once you're satisfied that your new code-coloring functions work correctly, you can then set the `recolor_range` and `recolor_from_here` variables to refer to them.

```
buffer int (*when_displaying)();
recolor_partial_code(int from, int to)
char first_window_refresh;
drop_all_colored_regions()
drop_coloring(int buf)
```

Epsilon calls the EEL subroutine pointed to by the buffer-specific function pointer `when_displaying` as it displays a window on the screen. It calls this subroutine once for each window, after determining which part of the buffer will be displayed, but before putting text for that window on the screen.

Epsilon sets the `first_window_refresh` variable prior to calling the `when_displaying` subroutine to indicate whether or not this is the first time a particular buffer has been displayed during a particular screen refresh. When a buffer appears in more than one window, Epsilon sets this variable to 1 before calling the `when_displaying` subroutine during the display of the first window, and sets it to zero before calling that subroutine during the display of the remaining windows. Epsilon sets the variable to 1 if the buffer only appears in one window. The value is valid only during a call to the buffer's `when_displaying` subroutine.

In a buffer with code coloring turned on, the `when_displaying` variable points to a subroutine named `recolor_partial_code()`. Epsilon passes two values to the subroutine that specify the range of the buffer that was modified since the last time the buffer was displayed. The standard `recolor_partial_code()` subroutine provided with Epsilon uses this information to discard any saved coloring data for the modified region of the buffer in the data structures it maintains. It then calls the two language-specific subroutines described at the beginning of this section as needed to color parts of the buffer.

The `drop_all_colored_regions()` subroutine discards coloring information collected for the current buffer. The next time Epsilon needs to display the buffer, it will begin coloring the buffer again. The `drop_coloring()` subroutine is similar, but lets you specify the buffer number. It also discards some data structures, so it's more suitable when the buffer is about to be deleted.

8.2.16 Colors

```
user int selected_color_scheme;
short _our_mono_scheme;
short _our_color_scheme;
short _our_gui_scheme;
short _our_unixconsole_scheme;
short *get_color_scheme_variable()
window short window_color_scheme;
```

Epsilon stores color choices in *color scheme* variables. A color scheme specifies the color combination to use for each defined color class.

Epsilon's standard color schemes are defined in the file stdcolor.e. See page 183 for the syntax of color definitions. You can also create additional color schemes without loading an EEL file by using the `new_variable()` primitive, providing `NT_COLSCHEME` as the second parameter. Epsilon stores color schemes in its name table, just like variables and commands, so a color scheme may not have the same name as a variable or other name table entry. (Color classes, on the other hand, have their own unique "name space".)

The `selected_color_scheme` primitive variable contains the name table index of the color scheme to use. Setting it changes the current color scheme. Each time Epsilon starts up, it sets this variable from one of four other variables: `_our_gui_scheme` under Epsilon for Windows or in Epsilon for Unix under X, `_our_unixconsole_scheme` if Epsilon for Unix is running in console mode and the `USE_DEFAULT_COLORS` environment variable is set (so that Epsilon inherits the colors of its xterm), `_our_mono_scheme` if Epsilon is running on a monochrome display, or `_our_color_scheme` otherwise. When you use set-color to select a different color scheme, Epsilon sets one of these variables, as well as `selected_color_scheme`. The `get_color_scheme_variable()` subroutine returns a pointer to one of these variables, the one containing a color scheme index that's appropriate for the current environment. By default, these four variables refer to the color schemes `standard-gui`, `xterm-color`, `standard-mono` and `standard-color`, respectively.

If the window-specific variable `window_color_scheme` is non-zero in a window, Epsilon uses its value in place of the `selected_color_scheme` variable when displaying that window. Epsilon uses this when displaying borderless windows, so that each window has an entirely different set of color class settings. Also see the variable `text_color`.

```
user char monochrome;
```

The `monochrome` variable is nonzero if Epsilon believes it is running on a monochrome display. Epsilon tries to determine this automatically, but the `-vmono` and `-vcolor` flags override this. See page 14.

```
set_color_pair(int colorclass, int fg, int bg, ?int scheme)
int get_foreground_color(int colorclass, ?int raw)
int get_background_color(int colorclass, ?int raw)
```

The `set_color_pair()` primitive lets you set the colors to use for a particular color class within the current color scheme (or, if the optional scheme argument is nonzero, in that scheme). The first parameter is a `color_class` expression (see page 176); the next two parameters are 32-bit numbers that specify the precise color to use. Use the `MAKE_RGB()` macro to construct suitable numbers. See page 183.

The `get_foreground_color()` and `get_background_color()` primitives let you retrieve the colors specified for a given color class. Normally they return a specific foreground or background color, after Epsilon has applied its rules for defaulting color specifications. (See page 183.) Specify a nonzero `raw` parameter, and Epsilon will return the color class's actual setting. It may include one of the bits `ETRANSPARENT`, `ECOLOR_COPY`, or `ECOLOR_UNKNOWN`.

The `ETRANSPARENT` macro is a special code that may be used in place of a background color. It tells Epsilon to substitute the background color of the `"text"` color class in the current color scheme. You can also use it for a foreground color, and Epsilon will substitute the foreground color of the `"text"` color class.

The `ECOLOR_UNKNOWN` macro in a foreground color indicates there's no color information in the current scheme for the specified color class.

The `ECOLOR_COPY` macro in a foreground color tells Epsilon that one color class is to borrow the settings of another. The index of other color class replaces the color in the lower bits of the value; use the `COLOR_STRIP_ATTR()` macro to extract it.

Regardless of whether the `raw` parameter is used, a retrieved foreground color may include any of the font style bits `EFONT_BOLD`, `EFONT_UNDERLINED`, or `EFONT_ITALIC`.

When Epsilon looks up the foreground and background settings of a color class, it uses this algorithm.

First it checks if the foreground color contains the `ECOLOR_UNKNOWN` code. If so, it tries to retrieve first a class-specific default, and then a scheme-specific default. First it looks for that color class in the `"color-defaults"` color scheme. This scheme is where Epsilon records all color class specifications that are declared outside any particular color scheme. If a particular color pair is specified as a default for that class, Epsilon uses that. If the color class has no default, Epsilon switches to the color class named `"default"` in the original color scheme and repeats the process.

Either the default setting for the color class or the original setting for the color class may use the `ECOLOR_COPY` macro. If so, then Epsilon switches to the indicated color class and repeats the above process. In the event that it detects a loop of color class cross-references or otherwise can't resolve the colors, it picks default colors.

Finally, if the resulting foreground or background colors use the `ETRANSPARENT` bit, Epsilon substitutes the foreground or background color from the `"text"` color class.

```
    int alter_color(int colorclass, int color)
    int rgb_to_attr(int rgb)
    int attr_to_rgb(int attr)
```

The `alter_color()` primitive is an older way to set colors. When the argument `color` is `-1`, Epsilon simply returns the color value for the specified color class. Any other value makes the color class use that color. Epsilon then returns the previous color for that color class. (In Epsilon for Windows or under the X windowing system, this function will return color codes, but ignores attempts to set colors. Use `set_color_pair()` to do this.)

The colors themselves (the second parameter to `alter_color()`) are specified numerically. Each number contains a foreground color, a background color, and an indication of whether blinking or extra-bright characters are desired.

The `alter_color()` function uses 4-bit color attributes to represent colors. The foreground color is stored in the low-order 4 bits of the 8-bit color attribute, and the background color is in the high-order 4 bits. Epsilon uses a pair of 32-bit numbers to represent colors internally, so `alter_color()` converts between the two representations as needed.

The functions `rgb_to_attr()` and `attr_to_rgb()` can be used to perform the same conversion. The `rgb_to_attr()` function takes a 32-bit RGB value and finds the nearest 4-bit attribute, using Epsilon's simple internal rules, while `attr_to_rgb()` converts in the other direction.

```
    int orig_screen_color()
```

In some environments, Epsilon records the original color attribute of the screen before writing text to it. The `orig_screen_color()` primitive returns this color code. If the `restore-color-on-exit` variable is nonzero, Epsilon sets the color class it uses after you exit (`color_class after_exiting`) to this color. See page 90.

```
    int number_of_color_classes()
    char *name_color_class(int colclass)
```

The `number_of_color_classes()` primitive returns the number of defined color classes. The `name_color_class()` primitive takes the numeric code of a color class (numbered from 0 to `number_of_color_classes() - 1`) and gives the name. For example, if the expression `color_class mode_line` has the value 3, then the expression `name_color_class(3)` gives the string `"mode-line"`.

Each window on the screen can use different color classes for its text, its borders, and its titles (if any). When a normal, tiled window is created, Epsilon sets its color selections from the color classes named `text`, `horiz_border`, `vert_border`, and `mode_line`. When Epsilon creates a pop-up window, it sets the window's color selections from the color classes `text`, `popup_border`, and `popup_title`. See page 90 for a description of the other predefined color classes.

```
    user window int text_color;
```

The `text_color` primitive contains the color class of normal text in the current window. You can get and set the other color classes for a window using the functions `get_wattrib()` and `set_wattrib()`.

8.3 File Primitives

8.3.1 Reading Files

```
int file_read(char *file, int transl)
```

The `file_read()` primitive reads the named file into the current buffer, replacing the text that was there. It returns an error code if an error occurred, or 0 if the read was successful. The `transl` parameter specifies the line translation to be done on the file. The buffer's `translation-type` variable will be set to its value. If `transl` is `FILETYPE_AUTO`, Epsilon will examine the file as it's read and set `translation-type` to an appropriate translation type.

```
int new_file_read(char *name, int transl,
                  struct file_info *f_info,
                  int start, int max)
```

The `new_file_read()` primitive reads a file like `file_read()` but provides more options. The `f_info` parameter is a pointer to a structure, which Epsilon fills in with information on the file's write date, file type, and so forth. The structure has the same format as the `check_file()` primitive uses (see page 257). If the `f_info` parameter is null, Epsilon doesn't get such information.

When Epsilon reads the file, it starts at offset `start` and reads at most `max` characters. You can use this to read only part of a big file. If `start` or `max` are negative, they are (individually) ignored: Epsilon starts at the beginning, or reads the whole file, respectively. The `start` parameter refers to the file before Epsilon strips ⟨Return⟩'s, while `max` counts the characters after stripping.

```
int do_file_read(char *s, int transl)  /* files.e */
buffer char _read_aborted;
int read_file(char *file, int transl)  /* files.e */
int find_remote_file(char *file, int transl)
file_convert_read(int flags)
do_readonly_warning()
update_readonly_warning(struct file_info *p)
```

Instead of calling the above primitives directly, extensions typically call one of several subroutines, all defined in files.e, that do things beyond simply reading in the file. Each takes the same two parameters as `file_read()`, and returns either 0 or an error code.

The `do_file_read()` subroutine records the file's date and time, so Epsilon can later warn the user that a file's been modified on disk, if necessary. If the user aborted reading the file, `do_file_read()` sets the `_read_aborted` variable nonzero. Epsilon then warns the user if he tries to save the partial file. This subroutine also handles reading URL's by calling the `find_remote_file()` subroutine, and character set translations such as OEM translations (see page 253) by calling `file_convert_read()`.

The `read_file()` subroutine calls `do_file_read()`, then displays either an error message, if a read error occurred, or the message "New file." It also handles calling `do_readonly_warning()` when it detects a read-only file, or `update_readonly_warning()` otherwise. (The latter can turn off a buffer's read-only attribute, if the file is no longer read-only.)

```
int find_in_other_buf(char *file, int transl)  /* files.e */
call_mode(char *file)                    /* files.e */
```

The `find_in_other_buf()` subroutine makes up a unique buffer name for the file, based on its name, and then calls `read_file()`. It then goes into the appropriate mode for the file, based on the file's extension, by calling the `call_mode()` subroutine. (See page 69.)

```
int find_it(char *fname, int transl)      /* files.e */
int look_file(char *fname)                /* buffer.e */
```

The `find_it()` subroutine first looks in all existing buffers for the named file, just as the find-file command would. If it finds the file, it simply switches to that buffer. (It also checks the copy of the file on disk, and warns the user if it's been modified.) If the file isn't already in a buffer, it calls `find_in_other_buf()`, and returns 0 or its error code. The `find_it()` subroutine uses the `look_file()` subroutine to search through existing buffers for the file. The `look_file()` subroutine, defined in buffer.e, returns 0 if no buffer has the file. Otherwise, it returns 1 and switches to the buffer by setting `bufnum`.

```
int do_find(char *file, int transl)         /* files.e */
```

Finally, the `do_find()` subroutine is at the top of this tree of file-reading functions. It checks to see if its "file name" parameter is a directory. If it is (or if it's a file pattern with wildcard characters), it calls `dired_one()` to run dired on the pattern. If it's a normal file, `do_find()` calls `find_it()`.

```
int err_file_read(char *file, int transl)      /* files.e */
```

Use the `err_file_read()` subroutine when you want to read a file that must exist, but you don't want all the extras that higher-level functions provide: checking file dates, choosing a buffer, setting up for read-only files, and so forth. It calls `file_read()` to read the file into the current buffer, and displays an error message if the file couldn't be read for any reason. It returns the error code, or 0 if there were no errors.

8.3.2 Writing Files

```
int file_write(char *file, int transl)
```

The `file_write()` primitive attempts to write the current buffer to the named file. It returns 0 if the write was successful, or an error code if an error occurred. The `transl` parameter specifies the line translation to be done while writing the file. See the description of `translation-type` below.

```
int new_file_write(char *name, int transl,
                   struct file_info *f_info,
                   int start, int max)
#define FILE_IO_ATSTART        -1
#define FILE_IO_NEWFILE        -2
#define FILE_IO_TEMPFILE       -3
char *get_tempfile_name()
```

The `new_file_write()` primitive writes a file, like `file_write()`, but provides more options. The `f_info` parameter is a pointer to a structure, which Epsilon fills in with information on the file's write date, file type, and so forth, just after it finishes writing the file. The structure has the same format as the `check_file()` primitive uses (see page 257). If the `f_info` parameter is null, Epsilon doesn't gather such information.

Different values for the `start` parameter change Epsilon's behavior. In the usual case, pass the value `FILE_IO_ATSTART`. That makes Epsilon open or create the file normally and start writing at its beginning, replacing its existing contents.

A `start` value of `FILE_IO_NEWFILE` makes the `new_file_write()` call fail if the file already exists.

A `start` value of `FILE_IO_TEMPFILE` makes Epsilon ignore the specified file name and pick an unused file name for a temporary file, in a directory designated for temporary files. The `get_tempfile_name()` primitive returns a pointer to the most recent temporary file created in this way (in a static buffer that will be reused for the next temporary file name).

With any of the above `start` codes, whatever Epsilon writes replaces the previous contents of the file. If `start` is greater than or equal to zero, though, Epsilon only rewrites a section of the existing file, starting at the offset specified by `start`, and the rest of the file's data will not change.

If the `max` parameter is non-negative, Epsilon writes only the specified number of characters. (Epsilon counts the characters before adding any ⟨Return⟩ characters or performing any encoding; the count is of characters in the current buffer.) If `max` is negative, Epsilon writes the entire buffer to the file.

```
int do_save_file(int backup, int checkdate,
                 int getdate) /* files.e */
```

The `do_save_file()` subroutine saves the current buffer like the save-file command, but lets you skip some of the things save-file does. Set the `backup` parameter to 0 if you don't want a backup file created, even if `want-backups` is nonzero. Set `checkdate` to 0 if you don't want Epsilon to check that the file on disk is unchanged since it was read. Set `getdate` to 0 if you don't want Epsilon to update its notion of the file's date, after the file has been written.

The function returns 0 if the write was successful, 1 if an error occurred, or 2 if the function asked the user to confirm a questionable write, and the user decided not to write the file after all.

```
int ask_save_buffer()
int warn_existing_file(char *s)
```

A command can call the `ask_save_buffer()` subroutine before deleting a buffer with unsaved changes. It asks the user if the buffer should be saved before it's deleted, and returns non-zero if the user asked that the buffer be saved. The caller is responsible for actually saving the file.

Before writing to a user-specified file, a command may call the `warn_existing_file()` subroutine. This will check if the file already exists and warn the user that it will be overwritten. The subroutine returns zero if the file didn't exist, or if the user said to go ahead and overwrite it, or nonzero if the user said not to overwrite it.

8.3.3 Line Translation

Epsilon normally deals with files with lines separated by the ⟨Newline⟩ character. Windows, DOS and OS/2, however, generally separate one line from the next with a ⟨Return⟩ character followed by a ⟨Newline⟩ character. For this reason, Epsilon normally removes all ⟨Return⟩ characters from a file when it's read from disk, and places a ⟨Return⟩ character before each ⟨Newline⟩ character when a buffer is written to disk, in these environments. But Epsilon has several other line translation methods:

The `FILETYPE_BINARY` translation type tells Epsilon not to modify the file at all when reading or writing.

The `FILETYPE_MSDOS` translation type tells Epsilon to remove ⟨Return⟩ characters when reading a file, and insert a ⟨Return⟩ character before each ⟨Newline⟩ when writing a file.

The `FILETYPE_UNIX` translation type tells Epsilon not to modify the file at all when reading or writing. It's similar to `FILETYPE_BINARY` (but Epsilon copies buffer text to the system clipboard in a different way).

The `FILETYPE_MAC` translation type tells Epsilon to convert ⟨Return⟩ characters to ⟨Newline⟩ characters when reading a file, and to convert ⟨Newline⟩ characters to ⟨Return⟩ characters when writing a file.

The `FILETYPE_AUTO` translation type tells Epsilon to examine the contents of a file as it's read, and determine the proper translation type using a heuristic. Epsilon then reads the file using that translation type, and sets `translation-type` to the new value. Normally this value is only used when reading a file, not when writing one. If you try to write a file and specify a translation type of `FILETYPE_AUTO`, it will behave the same as `FILETYPE_MSDOS` (except in Epsilon for Unix, where it's the same as `FILETYPE_UNIX`.

```
user buffer int translation_type;  /* EEL variable */
#define FORCED_TRANS            (8)
#define MAKE_TRANSLATE(e, t, f)  (((e) << 4) | ((t) & 7) \
                                 | ((f) ? FORCED_TRANS : 0))
#define GET_ENCODING(t)         ((t) >> 4)
#define GET_LINE_TRANSLATE(t)   ((t) & 7)
#define SET_TRANSLATE(t, trans)  (((t) & ~7) | ((trans) & 7))
#define SET_ENCODING(t, e)      (((t) & 15) | ((e) << 4))
```

The buffer-specific variable `translation-type` includes the current buffer's translation type as one of the above codes, combined with an encoding number that specifies the current buffer's encoding. Most functions for reading or writing a file take such a value as a `transl` parameter.

You can combine one of the above translation codes with an encoding number using the `MAKE_TRANSLATE()` macro. It takes a translation code `t`, an encoding code `e`, and a code `f` which, if nonzero, indicates that the resulting translation value was explicitly specified by the user in some way, not auto-detected.

Use the macros `GET_ENCODING()` and `GET_LINE_TRANSLATE()` to extract the encoding code or the line translation code, respectively, from a translation type value.

Use the macros `SET_ENCODING()` and `SET_TRANSLATE()` to combine an existing translation type value `t` with a new encoding code or line translation code, respectively.

```
user int default_translation_type;
user int new_buffer_translation_type;
int ask_line_translate()
```

A user can set the `default-translation-type` variable to one of the above translation codes to force Epsilon to use a specific translation when it reads an existing file. If this variable is set to its default value of `FILETYPE_AUTO`, Epsilon examines the file to determine a translation method. Setting this variable to any other value forces Epsilon to use that line translation method for all files.

When Epsilon creates a new buffer, it sets the buffer's `translation-type` variable to the value of the `new-buffer-translation-type` variable. Epsilon does the same when you try to read a file that doesn't exist. You can set this variable if you want Epsilon to examine existing files to determine their translation type, but create new files with a specific translation type. By default this variable is set to `FILETYPE_AUTO`, so the type for new buffers becomes `FILETYPE_UNIX` in Epsilon for Unix, and `FILETYPE_MSDOS` elsewhere.

The `ask_line_translate()` subroutine defined in files.e helps to select the desired translation method and encoding. Many commands that read a user-specified file call it. If a numeric prefix argument was not specified, it returns the value of the `default-translation-type` variable. But if a numeric prefix argument was specified, it prompts the user for the desired translation type and encoding.

8.3.4 Character Encoding Conversions

```
char *encoding_to_name(int enc)
int encoding_from_name(char *name)
```

When Epsilon reads or writes a file, it converts text between the Unicode character representation it uses internally and one of various file encodings. Epsilon represents each possible encoding with a number.

These numbers may change from one version of Epsilon to the next, so if an encoding setting must be recorded somehow, it should be recorded by name, not by number. Certain specific encodings will not change their codes: the encoding "auto-detect" is always numbered 0, and the encoding "raw" is always numbered 1.

The `encoding_from_name()` primitive returns the number of an encoding given its name. It returns −1 pointer if the encoding name is unknown.

The `encoding_to_name()` primitive returns the name of an encoding given its number. It returns a NULL pointer if the encoding number is unknown. Many encodings have more than one name; this primitive returns a particular standard one.

```
int file_convert_write(char *file, int trans,
                       struct file_info *f_info)
zeroed buffer char file_write_newfile;
int save_remote_file(char *fname, int trans,
                     struct file_info *finfo)
buffer char *(*file_io_converter)();
char *oem_file_converter(int func)
zeroed char *(*new_file_io_converter)();
```

The `do_save_file()` subroutine uses the `file_convert_write()` subroutine to actually write the file. Like `new_file_write()`, it takes a file name, a line translation code as described under `translation-type`, and a structure which Epsilon will fill with information on the file's write date, file type, and so forth. See `do_save_file()` above for details.

Unlike primitives such as `new_file_write()`, the `file_convert_write()` subroutine knows how to handle URL files by calling the `save_remote_file()` subroutine. It also checks the `file_write_newfile` variable; if nonzero, it ensures that writing fails if the specified file already exists.

In addition to the built-in conversion codes described above, Epsilon also supports user-defined EEL conversion routines. These are currently used only for DOS/OEM files read using the find-oem-file command and similar. The `file_convert_write()` subroutine handles writing these. It looks for a buffer-specific variable `file_io_converter`. This variable can be null, for no special translation, or it can contain a function pointer. For OEM files, for example, it points to the subroutine `oem_file_converter()`.

Any such subroutine will be called with a code indicating the desired action. The codes are defined in eel.h. The code `FILE_CONVERT_READ` tells the subroutine to translate the text in the current buffer as appropriate when reading a file. The code `FILE_CONVERT_WRITE` tells the subroutine to translate the buffer as appropriate when writing a file.

Before actually performing a conversion, Epsilon will call the subroutine to ask if the conversion is safe (reversible), by passing the `FILE_CONVERT_ASK` in addition to one of the above flags. A conversion is reversible, and therefore safe, if the conversion followed by the opposite conversion (for instance, ANSI => OEM => ANSI) yields the original text. If the conversion isn't safe, the subroutine should ask the user for permission to proceed.

The converter should then return a null pointer to cancel the read or write operation, or any other value to let it proceed. You can add the `FILE_CONVERT_QUIET` flag, and the converter won't ask the user for confirmation, merely return a value indicating whether the conversion would be safe.

Whenever the `FILE_CONVERT_ASK` flag isn't present, the subroutine should return the name of its minor mode—Epsilon will display this in the mode line. The OEM converter returns `" OEM"`.

When creating a new buffer, file-reading subroutines initialize the `file_io_converter` variable by copying the value of `new_file_io_converter`. Commands like find-oem-file temporarily set this variable to effect reading a file with OEM translation.

```
int perform_unicode_conversion(int buf, int from, int to,
                               int flags, char *encoding)
```

The `perform_unicode_conversion()` primitive converts between 16-bit Unicode characters and various 8-bit encodings such as UTF-8. It converts characters in the range `from`...`to` in the specified buffer `buf` in place.

By default, the primitive converts from 16-bit Unicode characters to the named 8-bit `encoding`. The `CONV_TO_16` flag makes it convert in the opposite direction, from the specified 8-bit encoding to 16-bit characters.

The primitive returns the code `EBADENCODE` if it doesn't recognize the encoding name. It returns `ETOOBIG` when converting from 8-bit characters if one of the characters is outside the range 0–255. It returns 0 on success. The primitive moves point to the end of the buffer.

If the specified encoding has a defined signature (a byte order mark), and an entire buffer was converted, not just part of one, Epsilon adds the signature when converting to the encoding, and removes the signature, if there is one, when converting from the encoding.

```
int buffer_flags(int buf)
```

Internally, Epsilon stores the text of a buffer with 8 bits for each character, unless it contains some characters outside the range 0–255. In that case it uses 16 bits for each character. A buffer that once contained such characters but no longer does may still be stored as 16 bits per character. Epsilon transparently handles all needed translations between the two formats (for instance, when you copy text from one buffer to another), but it's occasionally useful to tell which format Epsilon is using.

The `buffer_flags()` primitive returns a bit mask. Check the bit represented by the `BF_UNICODE` macro; if it's present, the specified buffer `buf` is stored in 16-bit format internally. If `buf` is omitted or zero, the primitive checks the current buffer.

8.3.5   More File Primitives

```
user buffer short modified;
int get_buf_modified(int buf)
set_buf_modified(int buf, int val)
int unsaved_buffers()
zeroed buffer char save_without_prompt;
int is_unsaved_buffer()
int buffer_unchanged()
```

Epsilon maintains a variable that tells whether the buffer was modified since it was last saved to a file. The buffer-specific variable `modified` is set to 1 each time the current buffer is modified. It is set to 0 by the

`file_read()`, `file_write()`, `new_file_read()`, and `new_file_write()` primitives, if they complete without error.

The `get_buf_modified()` and `set_buf_modified()` subroutines let you get or set the value of this variable for some other buffer `buf`.

The `unsaved_buffers()` subroutine defined in files.e returns 1 if there are any modified buffers. It doesn't count empty buffers, or those with no associated file names. If an EEL program creates a buffer that has an associated file name and is marked modified, but still doesn't require saving, it can set the buffer-specific variable `discardable_buffer` nonzero to indicate that the current buffer doesn't require any such warning. An EEL program can set the buffer-specific `save_without_prompt` variable nonzero for a buffer to have it silently saved without prompting, whenever Epsilon checks for unsaved buffers.

The `unsaved_buffers()` subroutine calls the `is_unsaved_buffer()` subroutine to check on an individual buffer. It tells if the current buffer shouldn't be deleted, and checks for the `discardable_buffer` variable as well as the `buffer-not-saveable` variable and other special kinds of buffers.

The `buffer_unchanged()` primitive returns a nonzero value if the current buffer has been modified since the last call of the `refresh()` or `maybe_refresh()` primitives. It returns zero if the buffer has not changed since that time. Epsilon calls `maybe_refresh()` to display the screen after each command.

```
user buffer char *filename;
set_buffer_filename(char *file)
```

The file reading and writing functions are normally used with the file name associated with each buffer, which is stored in the buffer-specific `filename` variable. To set this variable, use the syntax `filename = ` *new value*`;`. Don't use `strcpy()`, for example, to modify it.

The `set_buffer_filename()` subroutine defined in files.e sets the file name associated with the current buffer. However, unlike simply setting the primitive variable `filename` to the desired value, this function also modifies the current buffer's name to match the new file name, takes care of making sure the file name is in absolute form, and updates the buffer's access "timestamp". The bufed command uses this timestamp to display buffers sorted by access time.

```
user int errno;
file_error(int code, char *file, char *unknown)
char no_popup_errors;
```

File primitives that fail often place an error code in the `errno` variable. The `file_error()` primitive takes an error code and a file name and displays to the user a textual version of the error message. It also takes a message to print if the error code is unknown.

Under MS-Windows, the `file_error()` primitive pops up a message box to report the error. If EEL code sets the `no_popup_errors` variable nonzero, Epsilon will display such messages in the echo area instead, as it does under other operating systems.

```
int do_insert_file(char *file, int transl) /* files.e */
int write_part(char *file, int transl, int start, int end)
```

The `do_insert_file()` subroutine inserts a file into the current buffer, like the insert-file command. The `write_part()` subroutine writes only part of the current buffer to a file. Each displays an error message if the file could not be read or written, and returns either an error code or 0.

```
locate_window(char *buf, char *file)  /* buffer.e */
int buf_in_window(int bnum)
```

The `locate_window()` subroutine defined in window.e tries to display a given file or buffer by changing windows. If either of the arguments is an empty string `""` it will be ignored. If a buffer with the specified name or a buffer displaying the specified file is shown in a window, the subroutine switches to that window. Otherwise, it makes the current window show the indicated buffer, if any.

The `buf_in_window()` primitive finds a window that displays a given buffer, and returns its window handle. It returns −1 if no window displays that buffer.

```
int delete_file(char *file)
```

The `delete_file()` primitive deletes a file. It returns 0 if the deletion succeeded, and −1 if it failed. The `errno` variable has a code describing the error in the latter case.

```
int rename_file(char *oldfile, char *newfile)
```

The `rename_file()` primitive changes a file's name. It returns zero if the file was successfully renamed, and nonzero otherwise. The `errno` variable has a code describing the error in the latter case. You can use this primitive to rename a file to a different directory, but you cannot use it to move a file to a different disk.

```
int copyfile(char *oldfile, char *newfile)
```

The `copyfile()` primitive makes a copy of the file named `oldfile`, giving it the name `newfile`, without reading the entire file into memory at once. The copy has the same time and date as the original. The primitive returns zero if it succeeds. If it fails to copy the file, it returns a nonzero value and sets `errno` to indicate the error.

```
int make_backup(char *file, char *backupname)
```

The `make_backup()` primitive does whatever is necessary to make a backup of a file. It takes the name of the original file and the name of the desired backup file, and returns 0 if the backup was made. Otherwise, it puts an error code in `errno` and returns a nonzero number. The primitive may simply rename the file, if this can be accomplished without losing any special attributes or permissions the original file has. If necessary, Epsilon copies the original file to its backup file.

```
int get_file_read_only(char *fname)
int set_file_read_only(char *fname, int val)
int set_file_opsys_attribute(char *fname, int attribute)
```

The `get_file_read_only()` primitive returns 1 if the file `fname` has been set read-only, 0 if it's writable, or −1 if the file's read-only status can't be determined (perhaps because the file doesn't exist). The `set_file_read_only()` primitive sets the file `fname` read-only (if `val` is nonzero) or writable (if `val` is zero). It returns 0 if an error occurred, otherwise nonzero.

Under Unix, `set_file_read_only()` sets the file writable for the current user, group and others, as modified by the current umask setting (as if you'd just created the file). Other permission bits aren't modified.

The `set_file_opsys_attribute()` primitive sets the raw attribute of a file. The precise meaning of the attribute depends on the operating system: under Unix this sets the file's permission bits, while in other environments it can set such attributes as Hidden or System. The primitive returns nonzero if it succeeds. See the opsysattr member of the structure set by `check_file()` to retrieve the raw attribute of a file.

```
int is_directory(char *str)
int is_pattern(char *str)
```

The `is_directory()` primitive takes a string, and asks the operating system if a directory by that name exists. If so, `is_directory()` returns 1; otherwise, it returns 0. Also see the `check_file()` primitive on page 257, and the `is_remote_dir()` subroutine on page 264.

The `is_pattern()` primitive takes a string, and tells whether it forms a file pattern with wildcards that may match several files. It returns 2 if its file name argument contains the characters `*` or `?`. These characters are always wildcard characters and never part of a legal file name. The function returns 1 if its file name argument contains any of the following characters: left square-bracket, left curly-bracket, comma, or semicolon. These characters can sometimes be part of a valid file name (depending upon the operating system and file system in use), but are also used as file pattern characters in Epsilon. It returns 3 if the file name contains both types of characters, and it returns 0 if the file name contains none of these characters.

```
user char file_pattern_wildcards;
#define FPAT_COMMA              (1)
#define FPAT_SEMICOLON          (2)
#define FPAT_SQUARE_BRACKET     (4)
#define FPAT_CURLY_BRACE        (8)
#define FPAT_ALL        (FPAT_COMMA | FPAT_SEMICOLON \
              | FPAT_SQUARE_BRACKET | FPAT_CURLY_BRACE)
```

You can control which of the characters [] {} , ; Epsilon will consider a wildcard character in file patterns by setting the `file-pattern-wildcards` variable. This affects the `do_dired()`, `is_pattern()`, `file_match()`, `dired_standardize()`, `check_file()`, and `is_directory()` primitives. Each bit in the variable enables a different set of characters.

`FPAT_COMMA` enables the `,` character, `FPAT_SEMICOLON` enables the `;` character, `FPAT_SQUARE_BRACKET` enables recognizing [] sequences, and `FPAT_CURLY_BRACE` lets Epsilon recognize {} sequences. The default value enables all these characters.

8.3.6   File Properties

```
int check_file(char *file, ?struct file_info *f_info)
```

The `check_file()` primitive gets miscellaneous information on a file or subdirectory from the operating system. It returns codes defined by macros in codes.h. If its argument `file` denotes a pattern that may match multiple files, it returns `CHECK_PATTERN`. (Use the `file_match()` primitive described on page 318 to retrieve the matches.) If `file` names a directory or a file, it returns `CHECK_DIR` or `CHECK_FILE`, respectively. If `file` names a device, it returns `CHECK_DEVICE`. If `file` has the form of a URL, not a regular file, it returns `CHECK_URL`.

Under operating systems that support it, `check_file()` returns `CHECK_PIPE` for a named pipe and `CHECK_OTHER` for an unrecognized special file. Otherwise, it returns 0. If `f_info` has a non-null value, `check_file()` fills the structure it points to with information on the file or directory, except when it returns 0 or `CHECK_URL`. The structure has the following format (defined in eel.h):

```
struct file_info {      /* returned by check_file() */
        int fsize;      /* file size in bytes */
        int opsysattr;  /* system dependent attribute */
        int raw_file_date_high;
```

```
                        /* opsys-dependent date: high 32 bits */
          int raw_file_date_low; /* low 32 bits */
          short year;     /* file date: 1980-2099 */
          short month;    /* 1-12 */
          short day;      /* 1-31 */
          short hour;     /* 0-23 */
          short minute;   /* 0-59 */
          short second;   /* 0-59 */
          byte attr;      /* epsilon standardized attribute */
          byte check_type; /* file/directory/device code */
};
#define ATTR_READONLY   1
#define ATTR_DIRECTORY  2
```

The `check_type` member contains the same value as `check_file()`'s return code. The `attr` member contains two flags: `ATTR_READONLY` if the file cannot be written, or `ATTR_DIRECTORY` if the operating system says the file is actually a directory. The `opsysattr` member contains a raw attribute code from the operating system: the meaning of bits here depends on the operating system, and Epsilon doesn't interpret them. (See the `set_file_opsys_attribute()` primitive to set raw attribute codes for a file.)

Epsilon also provides the timestamp of a file, in two formats. The interpreted format (year, month, etc.) uses local time, and is intended to match the file timestamp shown in a directory listing. By contrast, in most cases the raw timestamp (in seconds) won't be affected by a change in time zones, the arrival of daylight savings time, or similar things, as the interpreted format will be. Under some operating systems Epsilon doesn't provide a raw timestamp; these two fields will be zero in that case.

For the second parameter to `check_file()`, make sure you provide a *pointer* to a `struct file_info`, not the actual structure itself. You can omit this parameter entirely if you only want the function's return value.

```
unique_filename_identifier(char *fname, int id[3])
unique_file_ids_match(int a[3], int b[3])
```

The `unique_filename_identifier()` primitive takes a file name and fills the `id` array with a set of values that uniquely describe it. Two file names with the same array of values refer to the same file. (This can happen under Unix due to symbolic or hard links.) If the primitive sets `id[0]` to zero, no unique identifier was found; comparisons between two file names, one or both of which return `id[0]==0`, must assume that the names might or might not refer to the same file. At this writing only Epsilon for Unix supports this feature; in other versions, `unique_filename_identifier()` will always set `id[0]` to zero.

The `unique_file_ids_match()` subroutine compares two `id` arrays from `unique_filename_identifier()`, returning nonzero if they indicate the two file names supplied to `unique_filename_identifier()` refer to the same file, and zero if they do not, or Epsilon cannot determine this.

```
int compare_dates(struct file_info *a,
                  struct file_info *b)
format_date(char *msg, int year, int month,
            int day, int hour, int minute,
            int second)
format_file_date(char *s, struct file_info *p)
```

The `compare_dates()` subroutine defined in filedate.e can be used to compare the dates in two `file_info` structures. It returns 0 if they have the same date and time, a negative number if `a` is dated earlier than `b`, or positive if `a` is dated later than `b`.

The `format_date()` subroutine takes a date and converts it to text form, using the format specified by the `date-format` variable. The `format_file_date()` subroutine takes a `file_info` structure and converts it to text form by calling `format_date()`.

```
int check_dates(int save)                    /* filedate.e */
```

The `check_dates()` subroutine defined in filedate.e compares a file's time and date on disk with the date saved when the file was last read or written. If the file on disk has a later date, it warns the user and asks what to do. Its parameter should be nonzero if Epsilon was about to save the file, otherwise zero. The function returns nonzero if the user said not to save the file.

The following example command uses `check_file()` to display the current file name and its date in the echo area.

```
#include "eel.h"

command show_file_date()
{
    struct file_info ts;

    if (check_file(filename, &ts))
        say("%s: %d/%d/%d", filename,
                    ts.month, ts.day, ts.year);
}
```

8.3.7   Low-level File Primitives

```
int lowopen(char *file, int mode)
```

The following primitives provide low-level access to files. The `lowopen()` primitive takes the name of a file and a mode code. It returns a "file handle" for use with the other primitives. The mode may be 0 for reading only, 1 for writing only, or 2 for both. If the file doesn't exist already, the primitive will return an error, unless you use mode 3. Mode 3 creates or empties the file first, and permits reading and writing.

```
int lowread(int handle, byte *buf, int count)
int lowwrite(int handle, byte *buf, int count)
```

The `lowread()` primitive tries to read the specified number of bytes, putting them in the byte array `buf`, and returns the number of bytes it was able to read. A value of 0 indicates the file has ended. The `lowwrite()` primitive tries to write the specified number of bytes from the byte array `buf`, and returns the number it was able to write. A return value different from `count` may indicate that the disk is full. See page 290 for functions to help translate between bytes and characters.

```
int lowseek(int handle, int offset, int mode)
int lowclose(int handle)
```

The `lowseek()` primitive repositions within the file. If the mode is 0, it positions to the *offset*th byte in the file, if 1 to the *offset*th byte from the previous position, and if 2 to the *offset*th byte from the end. The primitive returns the new offset within the file. Finally, the `lowclose()` primitive closes the file. All these routines return −1 if an error occurred and set `errno` with its code.

```
int lowaccess(char *fname, int mode)
#define LOWACC_R        4               /* file is readable. */
#define LOWACC_W        2               /* file is writable.  */
#define LOWACC_X        1               /* file is executable.  */
```

The `lowaccess()` primitive takes a file name and a code indicating whether the file's read access, write access or execute access should be tested, or zero if only the file's existence need be checked. It returns 0 if the file is accessible for the specified purpose (can be read, can be written, can be executed, exists), or −1 if not.

8.3.8   Directories

```
int getcd(char *dir)
int chdir(char *dir)
```

The `getcd()` primitive returns the current directory, placing it in the provided string. The format is B:\harold\work.

The `chdir()` primitive sets the current directory. (It sets the current drive as well if its argument refers to a drive. For example, invoking `chdir("A:\letters");` sets the current drive to A, then sets the current directory for drive A to \letters. `chdir("A:");` sets only the current drive.) The result for this primitive is 0 if the attempt succeeded, and −1 if it failed. The `errno` variable is set with a code showing the type of error in the latter case.

```
put_directory(char *dir)   /* files.e subr. */
int get_buffer_directory(char *dir)
```

The `put_directory()` subroutine copies the directory part of the file name associated with the current buffer into `dir`. Normally the directory name will end with a path separator character like '/' or '\'. If the current buffer has no associated file name, `dir` will be set to the empty string.

The `get_buffer_directory()` subroutine gets the default directory for the current buffer in `dir`. In most cases this is the directory part of the buffer's `filename` variable, but special buffers like dired buffers have their own rules. The subroutine returns nonzero if the buffer had an associated directory. If the buffer has no associated directory, the subroutine puts Epsilon's current directory in `dir` and returns 0.

```
user char *process_current_directory;
```

Epsilon stores the concurrent process's current directory in the `process_current_directory` variable. Setting this variable switches the concurrent process to a different current directory. To set this variable, use the syntax `process_current_directory = new value;`. Don't use `strcpy()`, for example, to modify it.

Under Windows 95/98/ME, Epsilon only transmits current directory information to or from the process when the process stops for console input. Under Windows NT/2000/XP, Epsilon tries to detect the process's current directory from EEL code and set this variable. See the variable `use-process-current-directory` for more details. Under Unix, Epsilon tries to retrieve the process's current directory whenever you access this variable, but setting it has no effect.

```
int mkdir(char *dir)
int rmdir(char *dir)
```

The `mkdir()` subroutine makes a new directory with the given name, and the `rmdir()` subroutine removes an empty directory with the given name. Each primitive returns 0 on success and −1 on failure, and sets `errno` in the latter case, as with `chdir()`.

```
get_customization_directory(char *dir)
```

When Epsilon starts, it locates its customization directory, as described on page 11. The `get_customization_directory()` primitive copies the name of this directory to `dir`.

Dired Subroutines

```
int dired_one(char *files)      /* dired.e */
int create_dired_listing(char *files)
int make_dired(char *files)
int do_remote_dired(char *files)
int do_dired(char *files)
int is_dired_buf()        /* dired.e */
```

The `dired_one()` subroutine takes a file name pattern as its argument and acts just like the dired command does, making a dired buffer, filling it and putting it in dired mode. It puts its pattern in a standard form and chooses a suitable buffer name, then calls the `create_dired_listing()` subroutine. This function prepares the buffer and displays suitable messages, then calls `make_dired()`.

The `make_dired()` subroutine handles FTP dired requests by calling `do_remote_dired()`, and passes local dired requests to the `do_dired()` primitive to fill the buffer with directory information.

Each of these routines takes a file name with wildcard characters such as * and ?, and inserts in the current buffer exactly what the dired command does (see page 110). Each returns 0 normally, and 1 if there were no matches.

By default, the `do_dired()` primitive ignores the abort key. To permit aborting a long file match, set the primitive variable `abort_file_matching` using `save_var` to tell Epsilon what to do when the user presses the abort key. See page 318 for details.

The `is_dired_buf()` subroutine returns 1 if the current buffer is a dired buffer, otherwise 0.

```
dired_standardize(char *files)
standardize_remote_pathname(char *files)
remote_dirname_absolute(char *dir)
drop_dots(char *path)
```

Sometimes there are several interchangeable ways to write a particular file pattern. For example, /dir1/dir2/* always makes the same list of files as /dir1/dir2/ or /dir1/dir2. The `dired_standardize()` primitive converts a dired pattern to its simplest form, in place. In the example, the last pattern is considered the simplest form.

The `standardize_remote_pathname()` subroutine is similar, but operates on FTP URL's. It calls several other subroutines to help.

The `remote_dirname_absolute()` subroutine converts a relative remote pathname to an absolute one in place. It performs an FTP operation to get the user's home directory, then inserts it into the given pathname.

The `drop_dots()` subroutine removes `.` and interprets `..` in a pathname, modifying it in place. It removes any `..` components at the start of a path.

```
detect_dired_format()
zeroed buffer char dired_format;
#define DF_UNIX     1
#define DF_SIMPLE   2
#define DF_OLDNT    3
#define DF_VMS      4
int get_dired_item(char *prefix, int func)
```

The dired command supports several different formats for directory listings. Besides the standard format it uses for local directory listings, as generated by the `do_dired()` primitive, it understands the directory listings generated by FTP servers that run on Unix systems (and the many servers on other operating systems that use the same format), as well as several others.

The `detect_dired_format()` subroutine determines the proper format by scanning a dired buffer, and sets the `dired_format` variable as appropriate. A value of 0 indicates the default, local directory format. The other values represent other formats.

Various subroutines in dired use the `get_dired_item()` subroutine to help locate format-specific functions or variables, to do tasks that depend on the particular format. The subroutine takes a prefix like "dired-isdir-" and looks for a function named `dired_isdir_unix()` (assuming the `dired_format` variable indicates Unix). It returns the name table index of the function it found, if there is one, or zero otherwise.

If its parameter `func` is nonzero, it looks only for functions; if zero, it looks only for variables. You can use an expression like `(* (int (*)()) i)()` to call the function (assuming `i` is the value returned by `get_dired_item()`), or an expression like `get_str_var(i)` to get the value of a variable given its index.

8.3.9 Manipulating File Names

```
absolute(char *file, ?char *dir)
relative(char *abs, char *rel, ?char *dir)
int is_relative(char *fname)
```

Because the current directory can change, through use of the `chdir()` primitive described above, Epsilon normally keeps file names in absolute pathname form, with all the defaults in the name made explicit. It converts a file name to the appropriate relative pathname whenever it displays the name (for example, in the mode line).

The `absolute()` primitive takes a pointer to a character array containing a file name. It makes the file name be an absolute pathname, with all the defaults made explicit. For example, if the default drive is B:, the current directory is /harold/papers, the `path_sep` variable is '\' and the 80 character array `fname` contains "proposal"; calling `absolute()` with the argument `fname` makes `fname` contain "B:\harold\papers\proposal".

The primitive `relative()` does the reverse. It takes a file name in absolute form and puts an equivalent relative file name in a character array. Unlike `absolute()`, which modifies its argument in place, `relative()` makes a copy of the argument with the changes. If the default drive is B:, the current directory is \harold and the 80 character array `abs` contains B:\harold\papers\proposal, calling `relative(abs, rel)`; puts "papers\proposal" in the string array `rel`. You can also get a relative file name by using the `%r` format specifier in any Epsilon primitive that accepts a printf-style format string.

The `relative()` and `absolute()` primitives each take an optional additional argument, which names a directory. The `absolute()` primitive assumes that any relative file names in its first argument are relative to the directory named by the second argument. (If the second argument is missing or null, the primitive assumes that relative file names are relative to the current directory.) Similarly, if you provide a third argument to the `relative()` primitive, it makes file names relative to the specified directory, instead of the current directory.

Note that in EEL string or character constants, the \ character begins an escape sequence, and you must double it if the character \ is to appear in a string. Thus the Windows file name \harold\papers must appear in an EEL program as the string `"\\harold\\papers"`.

The `is_relative()` primitive returns nonzero if the file name looks like a relative pathname, not an absolute pathname. (It's not intended for use with URL's.)

```
char *get_tail(char *file, ?int dirok)
```

The `get_tail()` primitive takes a string containing a file name and returns a pointer to a position in the string after the name of the last directory. For example, suppose that `file` is the string "/harold/papers/proposal". Then

```
get_tail(file, 0)
```

would return a pointer to "proposal". Since the pointer returned is to the original string, you can use this primitive to modify that string. Using the above example, a subsequent

```
strcpy(get_tail(file, 0), "sample");
```

would make `file` contain the string "/harold/papers/sample". The `dirok` argument says what to do with a file name ending with a separator character '\' or '/'. If `dirok` is nonzero the primitive returns a pointer to right after the final separator character. If `dirok` is zero, however, the primitive returns a pointer to the first character of the final directory name. (If `file` contains no directory name, the primitive returns a pointer to its first character when `dirok` is zero.)

```
char *get_extension(char *file)
```

The `get_extension()` primitive returns a pointer to the final extension of the file name given as its argument. For example, an invocation of

```
get_extension("text.c")
```

would return a pointer to the ".c" part, and `get_extension("text")` would return a pointer to the null character at the end of the string. Like `get_tail()`, you can use this primitive to modify the string.

```
int is_path_separator(int ch)
```

The `is_path_separator()` primitive tells if a character is one of the characters that separate directory or drive names in a file name. It returns 1 if the character is '\' or '/', 2 if the character is ':', otherwise 0. Under Unix, it returns 1 if the character is '/', otherwise 0.

```
user char path_sep;
```

The `path_sep` variable contains the character for separating directory names. It is '\' under Windows, '/' under Unix.

```
add_final_slash(char *fname)
drop_final_slash(char *fname)
```

The `add_final_slash()` primitive adds a path separator character like / or \ to the end of `fname`, if there isn't one already. The `drop_final_slash()` primitive removes the last character of `fname` if it's a path separator. These primitives never count : as a path separator.

```
abbreviate_file_name(char *file, int room)
```

The `abbreviate_file_name()` subroutine defined in disp.e modifies the filename `file` so it's no more than `room` characters long, by replacing sections of it with an ellipsis (`...`). If `file` is no more than `room` characters long to begin with, it won't be changed. Values of `room` less than 10 will be treated as 10.

```
int is_remote_file(char *fname)
char url_services[50] = "ftp|http|telnet|scp|ssh";
int is_remote_dir(char *fname)
```

The `is_remote_file()` primitive tells whether `fname` looks like a valid URL. It returns 1 if `fname` starts with a service name like ftp://, http://, or telnet://, or 2 if `fname` appears to be an Emacs-style remote file name like /hostname:filename. It uses the `url_services` variable to determine which service names are valid; this must be a series of |-separated names.

The `is_remote_dir()` subroutine is somewhat similar; it tries to determine if a `fname` refers to a remote directory (or a file pattern; either should be passed to dired). It knows that `fname` can't be a remote directory unless it's a valid URL with a service type of ftp://. But then it has to guess, since it doesn't want to engage in a slow FTP conversation before returning with an answer. It assumes that a URL that ends in a / character is a directory; it recognizes wildcard characters; it looks for the ~notation that indicates a home directory. If none of these indicates a directory name, it returns zero to indicate something other than a remote directory.

```
get_executable_directory(char *dir)
```

The `get_executable_directory()` function stores the full pathname of the directory containing the Epsilon executable into `dir`.

```
look_up_tree(char *res, char *file, char *dir, char *stop)
int is_in_tree(char *file, char *tree)   /* files.e subr. */
```

The `look_up_tree()` subroutine searches for `file` in the given directory `dir`, its parent directory, and so forth, until it finds a file named `file` or reaches the root directory. If it finds such a file, it returns nonzero and puts the absolute pathname of the file into the character array `res`. If it doesn't find a file with the given name, it returns zero and leaves `res` set to the last file it looked for. If `file` is an absolute pathname to begin with, it puts the same file name in `res`, and returns nonzero if that file exists. If `dir` is a null pointer, `look_up_tree()` begins at the current directory. If `stop` is non-null, the function only examines child directories of the directory `stop`. The function stops as soon as it reaches a directory other than `stop` or one of its subdirectories. This function assumes that all its parameters are in absolute pathname form.

The `is_in_tree()` subroutine returns nonzero if the pathname `file` is in the directory specified by `dir` or one of its subdirectories. Both of its parameters must be in absolute pathname form.

```
user char path_list_char;
```

The `path_list_char` variable contains the character separating the directory names in a configuration variable like EPSPATH. It is normally ';', except under Unix, where it is ':'.

```
build_filename(char *result, char *pattern, char *file)
```

The `build_filename()` subroutine constructs file names from name templates (see page 99). It copies `pattern` to `result`, replacing the various % template codes with parts of `file`, which it obtains by calling the primitives `get_tail()` and `get_extension()`. The `expand_string_template()` subroutine on page 290 provides a more generalized facility to do this.

```
int fnamecmp(char *f1, char *f2)        /* buffer.e */
int filename_rules(char *fname)
```

The `fnamecmp()` subroutine compares two file names like the `strcmp()` primitive, returning 0 if they're equal, a positive number if the first comes before the second, or a negative number otherwise. However, it does case-folding on the file names first if this is appropriate for the particular file systems.

The `filename_rules()` primitive asks the operating system if a certain file system is case-sensitive or case-preserving, and returns other information too. It takes the name of any file or directory (which doesn't have to exist) on the file system, and returns a code whose values are represented by macros defined in codes.h. See page 102 for more information on how Epsilon determines the appropriate code for each file system.

The `FSYS_CASE_IGNORED` code indicates a non-case-preserving file system like DOS. The `FSYS_CASE_PRESERVED` code indicates a case-preserving file system like NTFS or VFAT. The `FSYS_CASE_SENSITIVE` code indicates a case-sensitive file system like Unix. The `FSYS_CASE_UNKNOWN` code indicates that Epsilon couldn't determine anything about the file system.

The function also returns a bit flag `FSYS_SHORT_NAMES`, valid whenever any code but `FSYS_CASE_UNKNOWN` is returned, that indicates whether only 8+3 names are supported. Use the mask macro `FSYS_CASE_MASK` to strip off this bit: for example, the expression

```
(filename_rules(f) & FSYS_CASE_MASK) == FSYS_CASE_SENSITIVE
```

is nonzero if the file system is case-sensitive.

The primitive also may return a bit indicating the type of drive a file is located on, if Epsilon can determine this. `FSYS_NETWORK` indicates the file is on a different computer and is being accessed over a network. `FSYS_CDROM` indicates the file is on a CD-ROM disk. `FSYS_REMOVABLE` indicates the file is on a removable medium like a floppy disk or Zip disk. And `FSYS_LOCAL` indicates the file is on a local (non-network) hard disk. At most one of the these bits will be present.

Epsilon for Unix returns `FSYS_CASE_SENSITIVE` for all files, even if they happen to lie on a file system that might use different rules natively. It can't detect the type of drive a file is on either.

```
int ok_file_match(char *s)        /* complete.e */
```

The `ok_file_match()` subroutine checks a file name to see if the `ignore_file_extensions` variable should exclude it from completion. It returns 0 if the file name should be excluded, or 1 if the file name is acceptable.

```
char *lookpath(char *file, ?int curdir)
char *look_on_path(char *file, int flags, char *path, ?int skip)
```

The `lookpath()` primitive looks in various standard Epsilon directories for a readable file with the supplied name. As soon as Epsilon locates the file, it returns the file's name. If it can't find the file, it returns a null pointer. See page 10 for more information on Epsilon's searching rules. The `look_on_path()` primitive is similar, but you can specify the path to use, and it offers some additional flexibility. These primitives will be described together.

First (for either primitive), if the specified file name is an absolute pathname, Epsilon simply checks to see if the file exists, and returns its name if it does, or a null pointer otherwise.

Next, if you call `lookpath()` with its optional parameter `curdir` nonzero (or if you call `look_on_path()` with the flag `PATH_ADD_CUR_DIR`), Epsilon looks for the file in the current directory. If `curdir` is zero or omitted (or `PATH_ADD_CUR_DIR` isn't specified), Epsilon skips this step (unless the file name explicitly refers to the current directory, like ".\filename").

The `lookpath()` primitive next looks for the file in the directory containing the Epsilon executable, then (except in Epsilon for Unix) in the parent of that directory. If Epsilon's executable is in a directory with a name like c:\epsilon\bin, it searches for the file in c:\epsilon\bin, then in c:\epsilon. The -w4 and -w8 flags tell Epsilon to skip these two steps, respectively. For `look_on_path()`, you must specify the flag `PATH_ADD_EXE_DIR` to search in the executable's directory, and `PATH_ADD_EXE_PARENT` to search in its parent.

If the file still cannot be found, `lookpath()` then locates the EPSPATH, the configuration variable containing the list of directories for Epsilon to search in. (`look_on_path()` uses the path provided as a parameter.) Epsilon looks in each of the directories in that path for a file with the given name, returning the full pathname of the file if it finds it, and a null pointer otherwise. The path must use the appropriate syntax for a directory list: directory names separated by colons under Unix, or by semicolons in other environments. If there is no EPSPATH, `lookpath()` in Epsilon for Unix substitutes ~/.epsilon:/usr/local/epsilon*VER*:/usr/local/epsilon:/opt/epsilon (where *VER* is a version string like 10.02; other versions skip this step in that case.

By default, `look_on_path()` only searches for files with the specified name. Add the `PATH_PERMIT_DIRS` flag if you want it to also return directories with that name. With the `PATH_PERMIT_WILDCARDS` flag, you can use a file pattern like `*.c` as the file name. The primitive will return the first matching file name.

If you supply `look_on_path()` with an optional `skip` parameter of *n*, it will skip over the first *n* matches it finds (so long as its parameter is a relative pathname). You can use this to reject a file and look for the next one on a path.

```
convert_to_8_3_filename(char *fname)
```

Under Windows, the `convert_to_8_3_filename()` primitive modifies the given file name by converting all long file names in fname to their short "8.3" file name aliases. Each component of a short file name has no more than eight characters, a dot, and no more than three more characters. For example, the file name "c:\Windows\Start Menu\Programs\Windows Explorer.lnk" might be translated to an equivalent file name of "c:\Windows\STARTM~1\Programs\WINDOW~1.LNK". Non-Windows versions of Epsilon will not modify the file name.

8.3.10   Internet Primitives

```
int telnet_host(char *host, int port, char *buf)
telnet_send(int id, char *text)
do_telnet(char *host, int port, char *buf)
buffer int telnet_id;
int telnet_server_echoes(int id)
```

Epsilon provides various commands that use Internet FTP, Telnet and similar protocols. This section documents how some parts of this interface work.

First, Epsilon provides the primitives `telnet_host()` and `telnet_send()` for use with the Telnet protocol. The `telnet_host()` function establishes a connection to a host on the specified port, and using the indicated buffer. It returns an identification code. The `telnet_send()` function can use this code to send text to the host. To kill the telnet job, call `telnet_send()` and pass NULL as the text. Commands normally call the `telnet_host()` function through the `do_telnet()` subroutine, which records the telnet identification code in the buffer-specific `telnet_id` variable, and does other housekeeping tasks.

The `telnet_server_echoes()` primitive accepts a telnet identification code as above, and returns 1 if the server on that connection is currently set to echo characters sent to it, or 0 if it is not.

```
int finger_user(char *user, char *host, char *buf)
int http_retrieve(char *resource, char *host, int port,
                   char *auth, char *buf, int flags)
char *http_force_headers;
```

The `finger_user()` primitive uses the Finger protocol to retrieve information on a particular user (if the host is running a Finger server). It takes the user name, the host, and the name of a buffer in which to put the results.

The `http_retrieve()` primitive uses the HTTP protocol to retrieve a page from a web site. It takes a resource name (the final part of a URL), a host, port, an authorization string (for password-protected pages) and destination buffer name, plus a set of flags. The `HTTP_RETRIEVE_WAIT` flag tells the function not to return until the transfer is complete. Without this flag the function begins the transfer and lets it continue in the background. The `HTTP_RETRIEVE_ONLY_HEADER` flag tells the function to retrieve only the header of the web page, not the body. Without this flag Epsilon will retrieve both; the first blank line retrieved separates the two.

If the `http_force_headers` variable is non-null and non-empty, `http_retrieve()` uses its contents for the HTTP request it sends to the remote system. It should contain a complete, valid HTTP request.

```
int ftp_op(char *buf, char *log, char *host, int port,
           char *usr, char *pwd, char *file, int op)
int do_ftp_op(char *buf, char *host, char *port,
              char *usr, char *pwd, char *file, int op)
```

The `ftp_op()` primitive uses the FTP protocol to send or retrieve files or get directory listings. It takes the destination or source buffer name, the name of a log buffer, a host computer name and port number, a user name and password, a file name, and an operation code that indicates what function it should perform (see below).

The `do_ftp_op()` subroutine is similar to `ftp_op()`, but it chooses the name of an appropriate FTP Log buffer, instead of taking the name of one as a parameter. Also, it arranges for the appropriate `ftp_activity()` function (see below) to be called, arranges for character-coloring the log buffer, and initializes the `ftp_job` structure that Epsilon uses to keep track of each FTP job.

The `FTP_RECV` operation code retrieves the specified file and the `FTP_SEND` code writes the buffer to the specified file name. The `FTP_LIST` code retrieves a file listing from the host of files matching the specified file pattern or directory name. The `FTP_MISC` code indicates that the file name actually contains a series of raw FTP commands to execute after connecting and logging in, separated by newline characters. Epsilon will execute the commands one at a time.

You can combine one of the above codes with some bit flags that modify the operation. Use the `FTP_OP_MASK` macro to mask off the bit flags below and extract one of the operation codes above.

Normally `ftp_op()` returns immediately, and each of these operations is carried out in the background. Add the code `FTP_WAIT` to any of the above codes, and the subroutine will not return until the operation completes.

The `FTP_ASCII` bit flag modifies the `FTP_RECV` and `FTP_SEND` operations. It tells Epsilon to perform the transfer in ASCII mode. By default, all FTP operations use binary mode, and Epsilon performs any needed line translation itself. But this doesn't work on some host systems (VMS systems, for example). See the `ftp-ascii-transfers` variable for more information.

The `FTP_USE_CWD` bit flag modifies how Epsilon uses the file name provided for operations like `FTP_RECV`, `FTP_SEND`, and `FTP_LIST`. By default, Epsilon sends the file name to the host as-is. For example, if you try to read a file `dirname/another/myfile`, Epsilon sends an FTP command like `RETR dirname/another/myfile`. Some hosts (such as VMS) use a different format for directory names than Epsilon's dired directory editor understands. So with this flag, Epsilon breaks a file name apart, and translates a request to read a file such as `dirname/another/myfile` into a series of commands to change directories to `dirname`, then to `another`, and then to retrieve the file `myfile`. The `FTP_PLAIN_LIST` bit flag makes `FTP_LIST` operations send a `LIST` command; without it, they send a `LIST -a` command so the remote system includes hidden files. The `ftp-compatible-dirs` variable controls these bits.

```
int url_operation(char *file, int op)
```

The `url_operation()` subroutine parses a URL and begins an Internet operation with it. It takes the URL and an operation code as described above for `ftp_op()`. If the code is `FTP_RECV`, then the URL may indicate a service type of telnet://, http://, or ftp://, but if the code is `FTP_SEND` or `FTP_LIST`, the service type must be ftp://. It can modify the passed URL in place to put it in a standard form. It calls one of the functions `do_ftp_op()`, `http_retrieve()`, or `do_telnet()` to do its work.

```
ftp_misc_operation(char *url, char *cmd)
```

The `ftp_misc_operation()` subroutine uses the `do_ftp_op()` subroutine to perform a series of raw FTP commands. It takes an ftp:// URL (ignoring the file name part of it) connects to the host, logs in, and then executes each of the newline-separated FTP commands in `cmd`. Dired uses this function to delete or move a group of files.

```
buffer int (*when_net_activity)();
net_activity(int activity, int buf, int from, int to)
```

As Epsilon performs Internet functions, it calls an EEL function to advise it of its progress. The buffer-specific variable `when_net_activity` contains a function pointer to the function to call. Epsilon uses the value of this variable in the destination buffer (or, in the case of the `NET_LOG_WRITE` and `NET_LOG_DONE` codes below, the log buffer). If the variable is zero in a buffer, Epsilon won't call any EEL function as it proceeds.

The EEL function will always be called from within a call to `getkey()` or `delay()`, so it must save any state information it needs to change, such as the current buffer, the position of point, and so forth, using `save_var`. The subroutine `net_activity()` shown above indicates what parameters the function should take—there's not actually a function by that name.

The `activity` parameter indicates the event that just occurred. A value of `NET_RECV` indicates that Epsilon has just received some characters and inserted them in a buffer. The `buf` parameter tells which buffer is involved. The `from` and `to` values indicate the new characters. A value of `NET_DONE` means that the net job running in buffer `buf` has finished. The above are the only activity codes generated for HTTP, Telnet, or Finger jobs.

FTP jobs have some more possible codes. `NET_SEND` indicates that another block of text has been sent. In this case, `from` indicates that number of bytes sent already from buffer `buf`, and `to` indicates the total number of bytes to be sent. The code `NET_LOG_WRITE` indicates that some more text has been written to the log buffer `buf`, in the range `from...to`. Finally, the code `NET_LOG_DONE` indicates that the FTP operation has finished writing to the log buffer. It occurs right after a `NET_DONE` call on FTP jobs.

```
ftp_activity(int activity, int buf, int from, int to)
finger_activity(int activity, int buf, int from, int to)
telnet_activity(int activity, int buf, int from, int to)
buffer int (*buffer_ftp_activity)();
```

The file epsnet.e defines the `when_net_activity` functions shown above, which provide status messages and similar things for each type of job. The `ftp_activity()` subroutine also calls a subroutine itself, defined just like these functions, through the buffer-specific variable `buffer_ftp_activity`. The dired command uses this to arrange for normal FTP activity processing when retrieving directory listings, but also some processing unique to dired.

```
int gethostname(char *host, ?int method)
```

The `gethostname()` primitive sets `host` to the computer's host name and returns 0. If it can't for any reason, it returns 1 and sets `host` to "?".

The `method` parameter controls which type of host name Epsilon retrieves. By default, it uses the machine's locally-configured host name. A value of 3 makes it instead retrieve the host's fully qualified domain name using DNS. Values of 1 or 2 make it do this only under Windows or Unix, respectively. Retrieving the DNS name in some network configurations can cause on-demand auto-dialing or delays if the machine's DNS server isn't accessible.

Parsing URL's

```
prepare_url_operation(char *file, int op, struct url_parts *parts)
get_password(char *res, char *host, char *usr)
int parse_url(char *url, struct url_parts *p)
```

Several subroutines handle parsing URL's into their component parts. These parts are stored in a `url_parts` structure, which has fields for a URL's service (http, ftp, and so forth), host name, port, user name if any, password if any, and the "file name": the final part of a URL, that may be a file name, a web page name or something else. Since an empty user name or password is legal, but is different from an omitted one, there are also fields to specify if each of these is present.

The `prepare_url_operation()` subroutine parses a URL and fills one of these structures. It complains if it doesn't recognize the service name, or if the service is something other than FTP but the operation isn't reading. The operation code is one of those used with the `ftp_op()` subroutine described on page 267. For example, it complains if you try to perform an `FTP_LIST` operation with a telnet:// URL. It also prompts for a password if necessary, and saves the password for later use, by calling the `get_password()` subroutine.

The `get_password()` subroutine gets the password for a particular user/host combination. Specify the user and host, and the subroutine will fill in the provided character array `res` with the password. The first time it will prompt the user for the information; it will then store the information and return it without prompting in future requests. The subroutine is careful to make sure the password never appears in a state

file or session file. To discard a particular remembered password, pass NULL as the first parameter. The next time get_password() is asked for the password of that user on that host, it will prompt the user again.

The prepare_url_operation() subroutine calls the parse_url() subroutine to actually parse the URL into a url_parts structure. The latter returns zero if the URL is invalid, or nonzero if it appears to be legal.

```
int split_string(char *part1, char *cs, char *part2)
int reverse_split_string(char *part1, char *cs, char *part2)
```

The parse_url() subroutine uses two helper subroutines. The split_string() subroutine divides a string part1 into two parts, by searching it for one of a set of delimiter characters cs. It finds the first character in part1 that appears in cs. Then it copies the remainder of part1 to part2, and removes the delimiter character and the remainder from part1. It returns the delimiter character it found. If no delimiter character appears in part1, it sets part2 to "" and returns 0. The reverse_split_string() subroutine is almost identical; it just searches through part1 from the other end, and splits the string at the last character in part1 that appears in cs.

```
char *get_url_file_part(char *url, int sep)
```

The get_url_file_part() subroutine helps to parse URL's. It takes a URL and returns a pointer to a position within it where its file part begins. For example, in the URL http://www.lugaru.com/why-lugaru.html, the subroutine returns a pointer to the start of "why". If sep is nonzero, the subroutine instead returns a pointer to the / just before "why". If its parameter is not a URL, the subroutine returns a pointer to its first character.

8.3.11 Tagging Internals

This section describes how to add tagging support to Epsilon for other languages. Epsilon already knows how to find tags in C and EEL files, and in assembly languages files.

```
tag_suffix_ext()       /* example function */
tag_suffix_none()
tag_suffix_default()
tag_mode_c()
```

When Epsilon wants to add tags for a file, it first looks at the file's extension and constructs a function name of the form tag_suffix_*ext*(), where *ext* is the extension. It tries to call this function to tag the file. If the file has no extension, it tries to call tag_suffix_none().

If there is no function with the appropriate name, Epsilon looks for a function based on the current buffer's mode. It constructs a function name of the form tag_mode_*mode*(), where *mode* is the value of the major_mode variable in the current buffer. If there is no mode-based function either, Epsilon calls tag_suffix_default() instead.

Thus, to add tagging for a language that uses file names ending in .xyz, define a function named tag_suffix_xyz(). Or if such files (and perhaps files with other extensions) use mode named "Xyz", define a function named tag_mode_xyz(). In most cases, a mode-based name is more convenient.

```
add_tag(char *func, int pos)
```

The tagging function will be called with point positioned at the start of the buffer to be tagged. (Epsilon preserves the old value of point.) It should search through the buffer, looking for names it wishes to tag. To add a tag, it should call the subroutine `add_tag()`, passing it the tag name and the offset of the first character of the name within the file. You can use the tagging functions for C and assembler as examples to write your own tagging functions. They are in the source file tags.e.

The pluck-tag command uses a regular expression pattern to parse an identifier in the buffer. By default, it uses the pattern in the variable `tag-pattern-default`. A mode can define a variable like `tag-pattern-perl` or `tag-pattern-c` to make Epsilon use a different pattern. (For instance, the pattern for C mode says that identifiers can include :: to specify a class name.)

Epsilon constructs a variable name, like `tag-pattern-perl`, from the current mode's name. If a variable by that name exists, pluck-tag uses it in place of `tag-pattern-default`.

8.4   Operating System Primitives

8.4.1   System Primitives

```
char *getenv(char *name)
putenv(char *name)
char *verenv(char *name)
```

Use the `getenv()` primitive to return entries from the environment. The primitive returns a null pointer if no environment variable `name` exists. For example, after the Windows command "`set waldo=abcdef`", the expression `getenv("waldo")` will return the string "abcdef".

The `putenv()` primitive puts strings in the environment. Normally environment entries have the form "NAME=definition". This primitive manipulates Epsilon's copy of the environment, which is passed on to any program that Epsilon runs, but it doesn't affect the environment you get when you exit from Epsilon. The value of the argument to `putenv()` is evaluated later, when you actually invoke some other program from within Epsilon. For this reason, it is important that the argument to `putenv()` not be a local variable.

The `verenv()` primitive gets configuration variables. Epsilon for Windows looks in the system registry. Under Unix it retrieves the variables from the environment, like `getenv()`.

Regardless of the operating system, this primitive looks for alternate, version-specific forms of the specified configuration variable. For example, in version 7.0 of Epsilon, `verenv("MYVAR")` would return the value of a variable named MYVAR70, if one existed. If not, it would try the name MYVAR7. If neither existed, it would return the value of MYVAR (or a null pointer if none of these variables were found). See page 8 for complete information on configuration variables.

```
short opsys;
#define OS_DOS  1    /* DOS or Windows */
#define OS_OS2  2    /* OS/2 */
#define OS_UNIX 3    /* Unix */
```

The `opsys` variable tells which operating system version of Epsilon is running, using the macros shown above and defined in codes.h. The primitive returns the same value for DOS and Windows; see the next definition to distinguish these.

```
short is_gui;
#define IS_WIN32S  1    /* (not supported) */
#define IS_NT      2
#define IS_WIN95   3
#define IS_WIN31   4    /* 16-bit version always says this */
```

The `is_gui` variable lets an EEL program determine if it's running in a version of Epsilon that provides general-purpose dialogs. The variable is nonzero only in the Windows GUI version. The values `IS_WIN32S`, `IS_NT`, and `IS_WIN95` indicate that the 32-bit version of Epsilon is running, and occur when the 32-bit version runs under Windows 3.1, Windows NT/2000/XP, and Windows 95/98/ME, respectively. The 16-bit version of Epsilon for Windows always uses the value `IS_WIN31`, even if you happen to be running it under a 32-bit version of Windows.

```
short is_unix;
#define IS_UNIX_TERM   1
#define IS_UNIX_XWIN   2
```

The `is_unix` variable is nonzero if Epsilon for Unix is running. It's set to the constant `IS_UNIX_XWIN` if Epsilon is running as an X program, or `IS_UNIX_TERM` if Epsilon is running as a terminal program.

```
short is_win32;
#define IS_WIN32_GUI       1
#define IS_WIN32_CONSOLE   2
```

The `is_win32` variable is nonzero if a version of Epsilon for 32-bit Windows is running, either the GUI version or the Win32 console version. The constant `IS_WIN32_GUI` represents the former. The constant `IS_WIN32_CONSOLE` represents the latter.

```
int has_feature;
```

Epsilon provides the `has_feature` variable so an EEL function can determine which facilities are available in the current environment. Bits represent possible features. Often these indicate whether a certain primitive is implemented.

```
#define MAX_CHAR       65535
```

The `MAX_CHAR` macro indicates the largest character code that can appear in a buffer.

```
ding()
maybe_ding(int want)        /* disp.e */
user int want_bell;          /* EEL variable */
user short beep_duration;
user short beep_frequency;
```

The `ding()` primitive produces a beeping sound, usually called the bell. It is useful for alerting the user to some error. Instead of calling `ding()` directly, however, EEL commands should call the `maybe_ding()` subroutine defined in disp.e instead. It calls `ding()` only if the variable `want_bell` is nonzero, and its parameter is nonzero. Pass one of the `bell_on_` variables listed on page 94 as the parameter. The sound that `ding()` makes is controlled by the `beep-duration` and `beep-frequency` variables. See page 94.

```
int clipboard_available()
int buffer_to_clipboard(int buffer_number, int convert_newlines,
                        int clipboard_format)
int clipboard_to_buffer(int buffer_number, int convert_newlines,
                        int clipboard_format)
```

`FEAT_ANYCOLOR`	Epsilon can use all RGB colors, not just certain ones.
`FEAT_GUI_DIALOGS`	`display_dialog_box()` is implemented.
`FEAT_FILE_DIALOG`	`common_file_dlg()` is implemented.
`FEAT_COLOR_DIALOG`	`comm_dlg_color()` is implemented.
`FEAT_SEARCH_DIALOG`	`find_dialog()` is implemented.
`FEAT_FONT_DIALOG`	`windows_set_font()` is implemented.
`FEAT_SET_WIN_CAPTION`	`set_window_caption()` is implemented.
`FEAT_OS_PRINTING`	`print_window()` is implemented.
`FEAT_WINHELP`	`win_help_string()` and similar are implemented.
`FEAT_OS_MENUS`	`win_load_menu()` and similar are implemented.
`FEAT_ANSI_CHARS`	Does this system normally use ANSI fonts, not DOS/OEM?
`FEAT_EEL_RESIZE_SCREEN`	Does EEL code control resizing the screen?
`FEAT_INTERNET`	Are Epsilon's Internet functions available?
`FEAT_SET_FONT`	Can EEL set the font via variables?
`FEAT_MULT_CONCUR`	Does Epsilon support multiple concurrent processes?
`FEAT_DETECT_CONCUR_WAIT`	Can Epsilon learn that a concurrent process waits for input?
`FEAT_EEL_COMPILE`	`eel_compile()` is implemented.
`FEAT_LCS_PRIMITIVES`	`lcs()` and related are implemented.
`FEAT_PROC_SEND_TEXT`	`process_send_text()` is implemented.
`FEAT_UNICODE`	16-bit Unicode characters are supported.

Figure 8.1: Bits in the has-feature variable.

The `clipboard_available()` primitive tells whether Epsilon can access the system clipboard in this environment. It returns nonzero if the clipboard is available, or zero if not. Epsilon for Windows can always access the clipboard. Epsilon for Unix can access the clipboard when it runs as an X program.

The `buffer_to_clipboard()` primitive copies the indicated buffer to the clipboard. A `clipboard_format` of zero means use the default format; otherwise, it specifies a particular Windows clipboard format code. If `convert_newlines` is nonzero, Epsilon will add a ⟨Return⟩ character before each ⟨Newline⟩ character it puts on the clipboard. This is the normal format for clipboard text. If `convert_newlines` is zero, Epsilon will put an exact copy of the buffer on the clipboard.

The `clipboard_to_buffer()` primitive replaces the contents of the given buffer with the text on the clipboard. The `clipboard_format` parameter has the same meaning as above. If `convert_newlines` is nonzero, Epsilon will strip all ⟨Return⟩ characters from the clipboard text before putting it in the buffer.

```
signal_suspend()
```

In Epsilon for Unix, the `signal_suspend()` primitive suspends Epsilon's job. Use the shell's fg command to resume it. When Epsilon runs as an X program, this primitive minimizes Epsilon instead.

8.4.2 Window System Primitives

```
windows_maximize()
windows_minimize()
windows_restore()
windows_foreground()
```

```
int windows_state()
int screen_to_window_id(int screen)
```

In Epsilon for Windows, and in Unix under X, the `windows_maximize()`, `windows_minimize()`, and `windows_restore()` primitives perform the indicated action on the main Epsilon screen. The `windows_foreground()` primitive tries to make Epsilon the foreground window; it does nothing under X.

The `windows_state()` primitive returns a code indicating the state of Epsilon's main window. The value `WINSTATE_MINIMIZED` indicates the window has been minimized or iconified. The value `WINSTATE_MAXIMIZED` indicates the window has been maximized. Zero indicates the window is in some other state.

The `screen_to_window_id()` primitive returns the system's window id (for X) or window handle (for Windows) corresponding to a particular screen number.

```
int drag_drop_result(char *file)
drag_drop_handler()
do_resume_client()
short reject_client_connections;
```

Epsilon uses the `drag_drop_result()` primitive to retrieve the names of files dropped on an Epsilon window using drag and drop, after receiving the event key `WIN_DRAG_DROP`. Pass the primitive a character array big enough to hold a file name. The primitive will return a nonzero value and fill the array with the first file name. Call the primitive again to retrieve the next file name. When the function returns zero, there are no more file names.

Epsilon uses this same method to retrieve server messages or DDE messages. When such a message arrives from another program, Epsilon parses the message as if it were a command line and then adds each file name to its list of drag-drop results. Epsilon for Unix doesn't support file drag and drop or DDE, only server messages from another copy of Epsilon.

When Epsilon returns the `WIN_DRAG_DROP` key, it also sets some mouse variables to indicate the source of the files that can be retrieved through `drag_drop_result()`. It sets `mouse_screen`, `mouse_x`, `mouse_y`, and similar variables to indicate exactly where the files were dropped. If the message arrived via DDE or due to -add or -wait, then `mouse_screen` will be -1.

The `drag_drop_result()` primitive returns 2 to indicate -wait was used to send the file name; 1 otherwise. If -wait was used in a client instance of Epsilon, the `do_resume_client()` primitive may be used to signal waiting clients that the user has finished editing the desired file and they may now resume.

The `drag_drop_handler()` subroutine in mouse.e handles the `WIN_DRAG_DROP` key. Don't bind this key to a subroutine with a different name; Epsilon requires that the `WIN_DRAG_DROP` key be bound to a function named `drag_drop_handler()` for correct handling of drag-drop.

A function may set the `reject_client_connections` variable nonzero to keep Epsilon from accepting any files or other messages from other clients, via either server messages or DDE. (Files may still be dropped on Epsilon.) In particular, other instances of Epsilon that use the -add flag will not see an instance of Epsilon where this variable has been set nonzero.

```
int dde_open(char *server, char *topic)
int dde_execute(int conv, char *msg, int timeout)
int dde_close(int conv)
```

Epsilon provides some primitives that you can use to send a DDE Execute message to another program under Windows.

First call `dde_open()` to open a conversation, providing the name of a DDE server and the topic name. It returns a conversation handle, or 0 if it couldn't open the conversation for any reason.

To send each DDE message, call `dde_execute()`. Pass the conversation handle from `dde_open()`, the DDE Execute message text to send, and a timeout value in milliseconds (10000, the recommended value, waits 10 seconds for a response). The primitive returns nonzero if it successfully sent the message.

Finally, call `dde_close()` when you've completed sending DDE Execute messages, passing the conversation handle. It returns nonzero if it successfully closed the connection.

WinHelp Interface

```
int win_help_contents(char *file)
```

The `win_help_contents()` primitive displays the contents page of the specified Windows help file. If the `file` parameter is `""`, it uses Epsilon's help file, displaying help on Epsilon. The function returns a nonzero value if it was successful.

```
int win_help_string(char *file, char *key)
```

The `win_help_string()` primitive looks up the entry for `key` in the specified Windows help file. If the `key` parameter is `""`, it shows the list of possible keywords. If the `file` parameter is `""`, it uses Epsilon's help file, displaying help on Epsilon. The function returns a nonzero value if it was successful.

```
windows_help_from(char *file, int show_contents)
```

The `windows_help_from()` subroutine wraps the above two subroutines. If there's a suitable highlighted region, it calls `win_help_string()` to display help on the keyword text in the highlighted region. Otherwise, it either displays the help file's contents topic (if `show_contents` is nonzero), or the help file's keyword index. The `windows_help_from()` subroutine also handles tasks like displaying an error if the user isn't running Epsilon for Windows.

The Menu Bar

```
int win_load_menu(char *file)
win_display_menu(int show)
```

The `win_load_menu()` primitive makes Epsilon read the specified menu file (normally `gui.mnu`), replacing all previous menu definitions. See the comments in the `gui.mnu` file for details on its format. The `win_display_menu()` primitive makes Epsilon display its menu bar, when its `show` parameter is nonzero. When `show` is zero, the primitive makes Epsilon remove the menu bar from the screen.

```
int win_menu_popup(char *menu_name)
```

The `win_menu_popup()` primitive pops up a context menu, as typically displayed by the right mouse button. The menu name must match one of the menu tags defined in the file `gui.mnu`, usually the tag `"_popup"`.

```
invoke_menu(int letter)
```

The `invoke_menu()` primitive acts like typing Alt-*letter* in a normal Windows program. For example, `invoke_menu('e')` pulls down the Edit menu. `Invoke_menu(' ')` pulls down the system menu. And `invoke_menu(0)` highlights the first menu item, but doesn't pull it down, like tapping and releasing the Alt key in a typical Windows program. (Also see the variable `alt-invokes-menu`.)

The Tool Bar

```
toolbar_create()
toolbar_destroy()
toolbar_add_separator()
toolbar_add_button(char *icon, char *help, char *cmd)
```

Several primitives let you manipulate the tool bar. They only operate in the Windows GUI version. The `toolbar_create()` primitive creates a new, empty tool bar. The `toolbar_destroy()` primitive hides the tool bar, deleting its contents. The `toolbar_add_separator()` primitive adds a blank space between buttons to the end of the tool bar.

The `toolbar_add_button()` primitive adds a new button to the end of the tool bar. The `cmd` parameter contains the name of an EEL function to run. The `help` parameter says what "tool tip" help text to display, if the user positions the mouse cursor over the button. The `icon` parameter specifies which icon to use. In this version, it must be one of these standard names:

```
STD_CUT        STD_PRINTPRE     VIEW_DETAILS
STD_COPY       STD_PROPERTIES   VIEW_SORTNAME
STD_PASTE      STD_HELP         VIEW_SORTSIZE
STD_UNDO       STD_FIND         VIEW_SORTDATE
STD_REDOW      STD_REPLACE      VIEW_SORTTYPE
STD_DELETE     STD_PRINT        VIEW_PARENTFOLDER
STD_FILENEW    VIEW_LARGEICONS  VIEW_NETCONNECT
STD_FILEOPEN   VIEW_SMALLICONS  VIEW_NETDISCONNECT
STD_FILESAVE   VIEW_LIST        VIEW_NEWFOLDER
```

Run the commands show-standard-bitmaps or show-view-bitmaps to see what they look like. Run the command standard-toolbar to restore the original tool bar.

```
user char want_toolbar;
```

Epsilon uses the `want_toolbar` primitive variable to remember if the user wants a tool bar displayed, in versions of Epsilon which support this.

Printing Primitives

```
struct print_options {
    int flags;      // Flags: see below.
    int frompage;   // The range of pages to print.
    int topage;
    int height;
    int width;
};

/* Epsilon supports these printer flags. */
#define PD_SELECTION            0x00000001
#define PD_PAGENUMS             0x00000002
#define PD_PRINTSETUP           0x00000040

short select_printer(struct print_options *p)
page_setup_dialog()
```

In the Windows version of Epsilon, the `select_printer()` primitive displays a dialog box that lets the user choose a printer, select page numbers, and so forth. The flags and parameters are a subset of those of the Windows API function `PrintDlg()`. The primitive returns zero if the user canceled printing, or nonzero if the user now wants to print. In the latter case, Epsilon will have filled in the `height` and `width` parameters of the provided structure with the number of characters that can fit on a page of text using the selected printer.

The `page_setup_dialog()` displays the standard Windows page setup dialog, which you can use to set printer margins or switch to a different printer.

```
short start_print_job(char *jobname)
short print_eject()
short end_print_job()
```

After using the `select_printer()` primitive, an EEL program that wishes to print must execute the `start_print_job()` primitive. It takes a string specifying the name of this job in the print queue. The EEL program can then print one or more pages, ending each page with a call to `print_eject()`. After all pages have been printed, the EEL program must call `end_print_job()`.

```
short print_line(char *str, ?int scheme)
short print_window(int win)
int create_invisible_window(int width, int height, int buf)
```

To actually produce output, two primitives are available. The `print_line()` primitive simply prints the given line of text, and advances to the next line. It prints using the "text" color class in the current color scheme. If the optional parameter `scheme` is nonzero, Epsilon uses that color scheme instead.

The `print_window()` primitive prints the contents of a special kind of Epsilon window. The window must have been created by calling `create_invisible_window()`, passing it the desired dimensions of the window, in characters, and the buffer it should display. The `create_invisible_window()` primitive returns a window handle which can be passed to `print_window()`. An EEL program can move through the buffer, letting different parts of the buffer "show" in this window, to accomplish printing the entire buffer. The invisible window may be deleted using the `window_kill()` primitive once the desired text has been printed.

8.4.3 Timing

```
int time_ms()
time_begin(TIMER *t, int len)
int time_done(TIMER *t)
int time_remaining(TIMER *t)
```

The `time_ms()` primitive returns the time in milliseconds since some arbitrary event in the past. Eventually, the value resets to 0, but just when this occurs varies with the environment. In some cases, the returned value resets to 0 once a day, while others only wrap around after longer periods.

The `time_begin()` and `time_done()` primitives provide easier ways to time events. Both use the TIMER data type, which is built into Epsilon. The `time_begin()` primitive takes a pointer to a TIMER structure and a delay in hundredths of a second. It starts a timer contained in the TIMER structure. The `time_done()` primitive takes a pointer to a TIMER that has previously been passed to `time_begin()` and returns nonzero if and only if the indicated delay has elapsed. The `time_remaining()` primitive returns the number of hundredths of a second until the delay of the provided timer elapses. If the delay has already

elapsed, the function returns zero. You can pass `-1` to `time_begin()` to create a timer that will never expire; `time_remaining()` will always return a large number for such a timer, and `time_done()` will always return zero.

Also see the `delay()` primitive on page 285.

```
current_time_and_date(char *s)
```

The `current_time_and_date()` subroutine fills in the string `s` with the current time and date. It uses the format specified by the `date-format` variable.

```
struct time_info {
        short year;      /* file date: 1980-2099 */
        short month;     /* 1-12 */
        short day;       /* 1-31 */
        short hour;      /* 0-23 */
        short minute;    /* 0-59 */
        short second;    /* 0-59 */
        short hundredth;/* 0-99 */
        short day_of_week; /* 0=Sunday ... 6=Saturday */
};
time_and_day(struct time_info *t_info)
```

The `time_and_day()` primitive requests the current time and day from the operating system, and fills in the `time_info` structure defined above. The structure declaration also appears in eel.h.

Notice that the `time_and_day()` primitive takes a *pointer* to a structure, not the structure itself. Here is an example command that prints out the time and date in the echo area.

```
#include "eel.h"

command what_time()
{
    struct time_info ts;

    time_and_day(&ts);
    say("It's %d:%d on %d/%d/%d.", ts.hour, ts.minute,
                                    ts.month, ts.day, ts.year);
}
```

8.4.4   Calling DLL's (Windows Only)

```
int call_dll(char *dll_name, char *func_name,
             char *ftype, char *args, ...)
```

The `call_dll()` primitive calls a function in a Windows DLL. Epsilon can only call 32-bit DLL's. The `dll_name` parameter specifies the DLL file name. The `func_name` parameter specifies the name of the particular function you want to call.

The `ftype` parameter specifies the routine's calling convention. The character `C` specifies the C calling convention, while `P` specifies the Pascal calling convention. Most Windows DLL's use the Pascal calling convention, but any function that accepts a variable number of parameters must use the C calling convention.

The `args` parameter specifies the type of each remaining parameter. Each letter in `args` specifies the type of one parameter, according to the following table.

Character	Description	
L	unsigned long	DWORD
I	int	INT, UINT, HWND, most other handles
S	far char *	LPSTR
P	far void *	LPVOID
R	far void **	LPVOID *

The I character represents a 32-bit parameter, and is equivalent to L in this version. L, S, P, and R always represent 32-bit parameters.

S represents a null-terminated string being sent to the DLL. P is passed similarly, but Epsilon will not check the string for null termination. It's useful when the string is an output parameter of the DLL, and may not be null-terminated before the call, or when passing structure pointers to a DLL.

R indicates that a DLL function returns a pointer by reference. Epsilon will pass the pointer you supply (if any) and retrieve the result. Use this for DLL functions that require a pointer to a pointer, and pass the address of any EEL variable whose type is "pointer to ..." (other than "pointer to function").

Here's an example, using `call_dll()` to determine the main Windows directory:

```
#define GetWindowsDirectory(dir, size)  (is_gui == IS_WIN31 \
    ? call_dll("kernel.dll", "GetWindowsDirectory", \
              "p", "pi", dir, size) \
    : call_dll("kernel32.dll", "GetWindowsDirectoryA", \
              "p", "pi", dir, size))

char dir[FNAMELEN];

GetWindowsDirectory(dir, FNAMELEN);
say("The Windows directory is %s", dir);
```

A DLL function that exists in both 16-bit and 32-bit environments will usually be in different .dll files, and will often go by a different name. Its parameters will often be different as well. In particular, remember that a structure that includes int members will be a different size in the two environments. To write an EEL interface to a DLL function that takes a pointer to such a structure, you'll need to declare two different versions of the structure, and pass the correct one to the DLL function, if you want your EEL interface to work in both 16-bit and 32-bit environments.

After you call a function in a DLL, Epsilon keeps the DLL loaded to make future calls fast. You can unload a DLL loaded by `call_dll()` by including just the name of the DLL, and omitting the name of any function or parameters. For example, `call_dll("extras.dll");` unloads a DLL named extras.dll.

```
char *make_pointer(int value)
```

The `make_pointer()` primitive can be useful when interacting with system DLL's. It takes a machine address as a number, and returns an EEL pointer that may be used to access memory at that address. No error checking will be done on the validity of the pointer.

8.4.5 Running a Process

```
int shell(char *program, char *cline, char *buf)
```

The `shell()` primitive takes the name of an executable file (a program) and a command line, pushes to the program, and gives it that command line. The primitive returns the result code of the wait() system call, or −1 if an error occurred. In the latter case, the error number is in `errno`.

The first argument to `shell()` is the name of the actual file a program is in, including any directory prefix. The second argument to `shell()` is the command line to pass to the program.

If the first argument to `shell()` is an empty string `""`, Epsilon behaves differently. In this case, Epsilon runs the appropriate shell command processor. (Note that `""` is not the same as NULL, a pointer whose value is 0.) If the second argument is also `""`, Epsilon runs the shell interactively, so that it prompts for commands. Otherwise, Epsilon makes the shell run only the command line specified in the second argument. Epsilon knows what flags to provide to the various standard shells to make them run interactively, or execute a single command and return, but you can set these if necessary. You can also set the command processor Epsilon should use. See page 119.

When you enter a DOS command outside of Epsilon, the command processor searches for the command by appending .com or .exe and looking in various directories for it, but Epsilon does not provide this service itself under DOS. However, the command processor itself provides a way to perform this search. Simply execute the command processor instead of the desired program by using `""` as the first argument, and prepend the name of the program (not including .com or .exe) to the desired command line. The command processor will then search for the program as usual. For example, to get the same result as typing

```
comp file1 file2
```

to the DOS command processor, use

```
shell("", "comp file1 file2", "");
```

instead of

```
shell("\\programs\\comp.com", "file1 file2", "");
```

(Note that you have to double the \ character if you want it to appear in an EEL string.) This technique is also necessary to execute batch files, use internal commands like "dir", or do command-line redirection.

The third argument to `shell()` controls whether the output of the program is to be captured. If `""`, no capturing takes place. Otherwise the output is inserted in the specified buffer, replacing its previous contents.

In Epsilon for Windows, when all three arguments to `shell()` are `""`, Epsilon starts the program and then immediately continues without waiting for it to finish. If any argument is nonempty, or in other versions, Epsilon waits for the program to finish.

```
int do_push(char *cmdline, int cap, int show)
```

The `do_push()` subroutine is a convenient way to call `shell()`. It uses the command processor to execute a command line (so the command line may contain redirection characters and the like). If cap is nonzero, the subroutine will capture the output of the command to the process buffer. If show is nonzero, the subroutine will arrange to show the output to the user. How it does this depends on `cap`. To show captured output, Epsilon displays the process buffer after the program finishes. To show non-captured output, Epsilon (non-GUI versions only) waits for the user to press a key after the program finishes, before restoring Epsilon's screen. If show is −1, Epsilon skips this step.

This subroutine interprets the variable `start-process-in-buffer-directory` and takes care of displaying an error to the user if the process couldn't be run.

Concurrent Process Primitives

```
int concur_shell(char *program, char *cline,
                  ?char *curdir, char *buf)
short another_process();
int is_process_buffer(int buf)
```

The `concur_shell()` primitive also takes a program and a command line, with the same rules as the `shell()` primitive. It starts a concurrent process, with input and output connected to the buffer "process", just like the start-process command described on page 119 does. If you specify a buffer `buf`, it starts the process in that buffer. (Some versions of Epsilon support only one process buffer; in them the buffer name, if specified, must be "process".) If you specify a directory name in `curdir`, Epsilon starts the process with that current directory. The primitive returns 0 if it could start the process. If it couldn't, it returns an error code.

Epsilon only receives concurrent process output and sends it input when Epsilon is waiting for you to press a key (or during a `delay()`—see page 285), but the process otherwise runs independently.

The `another_process()` primitive returns the number of active concurrent processes.

The `is_process_buffer()` primitive returns `ISPROC_CONCUR` if the specified buffer holds an active concurrent process, `ISPROC_PIPE` if the `buf_pipe_text()` primitive is sending output into it, or 0 if no concurrent process is associated with that buffer.

```
user buffer int type_point;
```

Characters from the process go into the process buffer at a certain position that we call the *type point*. The `type_point` variable stores this position.

When a process tries to read a character of input, Epsilon stops the process until there is at least one character following the type point, and when the process tries to read a line of input, Epsilon does not run the process until a newline appears in the section of the buffer after the type point. When a concurrent process is started by the `concur_shell()` primitive, the type point is initially set to the value of point in the specified buffer.

Internet commands for Telnet and FTP use `type_point` much like a process buffer does, to determine where to insert text into a buffer and where to read any text to be sent.

```
int process_input(?int buf)
#define PROCESS_INPUT_LINE  1
#define PROCESS_INPUT_CHAR  2
buffer int (*when_activity)();
concur_handler(int activity, int buf, int from, int to)
```

The `process_input()` primitive returns `PROCESS_INPUT_LINE` if the process is waiting for a character, `PROCESS_INPUT_CHAR` if the process is waiting for a line of input, and 0 if the process is running or there is no process. It operates on the buffer named "process" if no buffer number is specified.

Whenever Epsilon receives process output or sends it input, it calls an EEL function. The buffer-specific `when_activity` variable contains a function pointer to the function to call. If the variable is zero in a buffer, Epsilon won't call any EEL function as it proceeds. For a typical process buffer, the `when_activity` variable points to the `concur_activity()` subroutine.

Just after a concurrent process inserts output in a process buffer, it calls this subroutine, passing `NET_RECV` as the `activity`. The `from` and `to` parameters mark the range of buffer text that was just

received from the process. The `concur_activity()` subroutine responds to this message by coloring the inserted characters with the `color_class process_output` color, and similar tasks.

Epsilon calls this subroutine and passes `NET_SEND` when it detects that the concurrent process is now ready for input, and again as it sends the input to the process. When the process becomes ready for input, the subroutine will be called with a `from` parameter of zero. When the process is sent a line of text, the subroutine will be called with a `from` of `PROCESS_INPUT_LINE`, and when the process is sent a single character it will be called with a `from` of `PROCESS_INPUT_CHAR`. In each case the `to` parameter will indicate the beginning of the input text (the value of `type_point` before the input begins).

Epsilon calls this subroutine and passes `NET_DONE` when the process exits. Its `from` parameter will hold the exit code, or 0 if Epsilon didn't record this. Epsilon sets the buffer-specific `process_exit_status` variable to the value `PROC_STATUS_RUNNING` when a process starts, and sets it to the process exit status (or 0) when the process exits.

Epsilon for Unix often cannot detect when a process is awaiting input. Therefore `process_input()` always returns zero, and a `NET_SEND` activity will typically not be signaled with a `from` parameter of zero.

```
int process_send_text(int buf, char *text, int len)
```

Normally input to a process running in a concurrent process buffer comes from text the user inserts into the buffer. The `process_send_text()` primitive provides a way to send text directly to the process, bypassing the buffer. This is especially useful for passwords, since if a password appears in the buffer it might be seen, or retrieved with undo. The primitive sends `len` characters from `text` to the process associated with the buffer `buf`.

The `FEAT_PROC_SEND_TEXT` bit of the `has_feature` variable indicates when this primitive is available.

```
int halt_process(?int hard_kill, int buf)
```

The `halt_process()` primitive has the same function as the **stop-process** command. A value of `0` for `hard_kill` makes the primitive act the same as **stop-process** with no argument. Otherwise, it is equivalent to **stop-process** with an argument. The function returns `1` if it succeeds, and `0` if it cannot signal the process for some reason. It operates on the buffer named "process" if no buffer number is specified.

```
int process_kill(?int buf)
```

The `process_kill()` primitive disconnects Epsilon from a running concurrent process, telling it to exit. The function returns `1` if it succeeds, and `0` if it cannot kill the process for some reason. It operates on the buffer named "process" if no buffer number is specified.

Other Process Primitives

```
int pipe_text(char *input, char *output, char *cmdline,
              char *curdir, int flags, int handler)
my_handler(int activity, int buf, int from, int to)  // Sample.
int buf_pipe_text(int inputb, int outputb, char *cmdline,
                  char *curdir, int flags, ?int errorb)
```

The `pipe_text()` subroutine runs the program specified by `cmdline`, passing it the contents of a buffer as its standard input, and inserting its standard output into a second buffer (or the same buffer).

The input buffer name may be NULL if the process does not require any input. Epsilon provides a current directory of `curdir` to the process. It passes Epsilon's current directory if `curdir` is NULL or `""`. This subroutine returns `0` and sets `errno` if the function could not be started, or returns `1` if the function started successfully.

The `PIPE_SYNCH` flag means don't return from the subroutine until the process has finished. Without this flag, Epsilon starts the subprocess and then returns from `pipe_text()`, letting the subprocess run asynchronously.

The `PIPE_CLEAR_BUF` flag means empty the output buffer before inserting the process's text (but do nothing if the process can't be started); it's convenient when the input and output buffers are the same, to filter a buffer in place.

The `PIPE_NOREFRESH` flag tells Epsilon not to refresh the screen each time more data is received from the process, and is most useful with `PIPE_SYNCH` if you don't want the user to see the data until after it's been postprocessed in some way.

The `PIPE_SKIP_SHELL` flag makes Epsilon directly invoke the specified program, instead of using a shell as an intermediary. This results in improved performance, but command lines that use shell meta characters (like `>file` for redirection, `|` for pipelines, or file pattern wildcards) won't operate as desired. Only Epsilon for Unix supports this flag. When Epsilon prepares an argument list from the command line, it interprets and removes quotes which may surround arguments that contain spaces.

If `handler` is nonzero, it's the index of a function (that is, an EEL function pointer) to call each time text is received from the process, and when the process terminates. The handler function will be called with the buffer number into which more process output has just been inserted, and `from` and `to` set to indicate the new text. The parameter `activity` will be `NET_RECV` when characters have been received, or `NET_DONE` when the subprocess has exited. In the latter case `from` will hold the process exit code.

Epsilon sets the buffer-specific `process-exit-status` variable in the output buffer to the value `PROC_STATUS_RUNNING` when a process starts, and sets it to the process exit status (or 0) when the process exits.

The `pipe_text()` subroutine described above is implemented using the `buf_pipe_text()` primitive. There are a few differences between these:

The `buf_pipe_text()` primitive uses buffer numbers, not buffer names. It won't create a buffer for you the way the subroutine will; the buffer must already exist. (Pass `0` for a buffer number if you don't need input.)

Instead of passing a function pointer for `handler`, you must instead set the buffer-specific `when_activity` variable in the output buffer prior to calling `buf_pipe_text()`.

Pass a `curdir` of `""`, not NULL, to `buf_pipe_text()` to use Epsilon's current directory.

The `buf_pipe_text()` primitive accepts an additional, optional, parameter `errorb`. If nonzero, any output of the program sent to standard error will be sent to the `errorb` buffer instead of the `outputb` buffer. If `errorb` is zero, such output will appear in `outputb` along with standard output.

```
int winexec(char *prog, char *cmdline, int show, int wait)
/* Pass these values to winexec: */
#define SW_HIDE             0
#define SW_SHOWNORMAL       1
#define SW_SHOWMINIMIZED    2
#define SW_SHOWMAXIMIZED    3
#define SW_SHOWNOACTIVATE   4
```

```
#define SW_SHOW            5
#define SW_MINIMIZE        6
#define SW_SHOWMINNOACTIVE 7
#define SW_SHOWNA          8
#define SW_RESTORE         9
```

In Epsilon for Windows, the `winexec()` primitive runs a program, like the `shell()` primitive, but provides a different set of options. Normally, the second parameter to `winexec()` contains the command line to execute and the first parameter contains the name of the program to execute. With some versions of Windows and some types of executables, you can provide `""` as the program to execute, and Windows will determine the correct program name from the command line.

The third parameter to `winexec()` specifies the window visibility state for the new program. It can be one of the values listed above. If the fourth parameter is nonzero, Epsilon will wait for the program to finish before returning from the `winexec()` primitive. If the fourth parameter is zero, the primitive will return immediately.

This primitive returns the exit code of the program it ran. If an error prevented it from running the program, it returns −1 and puts an error code in the global variable `errno`. When the primitive runs a program without waiting for it to finish, the primitive returns zero if the program started successfully.

```
int run_viewer(char *file, char *action, char *dir)
```

The `run_viewer()` primitive runs the program associated with the given file, using its Windows file association. The most common `action` is `"Open"`, though a program may define others, such as `"Print"`. The `dir` parameter specifies the current directory in which to run the program. The primitive returns nonzero if it was successful, or zero if it could not run the program or the program returned an error code. This primitive always returns zero in the non-Windows versions of Epsilon.

8.5   Control Primitives

8.5.1   Control Flow

```
error(char *format, ...)
when_aborting()        /* control.e */
quick_abort()
```

Epsilon provides several primitives for altering the flow of control from one statement to the next. The `error()` primitive takes arguments like `say()`, displays the string as `say()` does, and then aborts the current command, returning to the main loop (see page 324). In addition this primitive discards any type-ahead and calls the user-defined subroutine `when_aborting()` if it exists. The standard version of `when_aborting()` optionally rings the bell and removes the erroneous command from any keyboard macro being defined. The primitive `quick_abort()` acts like `error()` but displays no message.

```
user char user_abort;
int abort_key;
check_abort()
```

The variable `user_abort` is normally 0. It is set to 1 when you press the key whose value is `abort_key`. To disable the abort key, set `abort_key` to −1. By default, the `abort_key` variable is set to Control-G. Use the set-abort-key command to set the `abort_key` variable. See page 84.

The primitive `check_abort()` calls `error()` with the argument `"Canceled."` if the variable `user_abort` is nonzero. Use the primitive `check_abort()` whenever a command can be safely aborted, since otherwise an abort will only happen when the command returns. Epsilon calls `check_abort()` internally during any searching operation (see page 204), when you use the `delay()` primitive (described below) to wait, or (optionally) during certain file matching primitives (see page 318).

```
leave(?int exitcode)
when_exiting()              /* EEL subroutine */
```

The primitive `leave()` exits Epsilon with the specified exit code (or 0 if omitted). A nonzero exit code keeps Epsilon from saving any settings it normally would, such as the currently selected font's name, or the sizes of any resized dialogs.

Just before calling `leave()`, Epsilon's standard commands call the `when_exiting()` subroutine. By default, this does nothing, but you can replace it to customize Epsilon's behavior at this time. (See page 293 to make sure your extension doesn't interfere with other extensions.)

```
delay(int hundredths, int condition, ?int buf)
```

The `delay()` primitive takes an argument specifying a period of time, in hundredths of a second, and a bit pattern specifying additional conditions (with codes specified in codes.h). It waits until one of the conditions occurs, or until the specified time limit is reached. A time limit of `-1` means to wait forever.

The condition code `COND_KEY` makes Epsilon return when a key is pressed or any key-generating input event occurs (like a mouse event, or getting the focus). The condition code `COND_TRUE_KEY` is similar, but only returns on actual keys, not mouse events or other events. The condition code `COND_PROC` makes Epsilon return when a concurrent process is waiting for input, or has exited. The condition code `COND_PROC_EXIT` makes Epsilon return when a concurrent process has exited. For the last two conditions, Epsilon checks on the buffer specified by the optional parameter `buf`. If `buf` is missing or zero, it checks the buffer named "process". These conditions are ignored if no process is running in the specified buffer.

The condition flag `COND_RETURN_ABORT`, in combination with `COND_KEY`, makes the `delay()` primitive return if the user presses the abort key, instead of aborting by calling the `check_abort()` primitive. (Note that if you don't specify `COND_KEY` or `COND_TRUE_KEY` as well, the primitive ignores all keys, including the abort key.)

This function varies a bit from one operating system to another. For example, the Unix version of Epsilon can't detect when a process is currently waiting for input, so it can only return when a process exits. Also see the timing functions on page 277.

```
int do_recursion()
leave_recursion(int val)
int recursive_edit() /* control.e */
char _recursion_level;
```

The `do_recursion()` primitive starts a new loop for getting characters and interpreting them as commands. A recursive edit preserves the current values of the variables `has_arg`, `iter`, `this_cmd`, and `prev_cmd`, but does not preserve the current buffer, window, or anything else. (See page 324.) Exit the recursion by calling the `leave_recursion()` primitive. It arranges for the main loop to exit, instead of waiting for another key to be executed. The call to `do_recursion()` will then return with a value of `val`, the argument of the call to `leave_recursion()`.

Sometimes a recursive edit is done "secretly," and the user doesn't know that one is being used. For example, when Epsilon reads the name of a file using completion, it's actually doing a recursive edit. Keys

like ⟨Space⟩ exit the recursive edit with a special code, and the function that did the recursive edit displays a menu, or whatever is needed, and then does another recursive edit.

Other times (typing Ctrl-R in query-replace, for example), the user is supposed to exit the recursive edit explicitly using the exit-level command. When you're supposed to use exit-level to exit, Epsilon displays extra [ ]'s in the mode line as a reminder. The `recursive_edit()` subroutine does a recursive edit, and arranges for these [ ]'s to appear by modifying the `_recursion_level` variable. It contains the number of extra [ ]'s to display. The `recursive_edit()` subroutine returns the value returned by `do_recursion()`.

If you call `leave_recursion()` when there has been no matching `do_recursion()`, Epsilon automatically invokes the command exit. If exit returns instead of calling the primitive `leave()`, Epsilon begins its main loop again.

```
int setjmp(jmp_buf *location)
longjmp(jmp_buf *location, int value)
```

Epsilon implements aborting by two special primitives that allow jumping from a function to another point in that function or any of the functions that called it. The `setjmp()` primitive marks the place to return, storing the location in a variable declared like this:

```
jmp_buf location;
```

After calling `setjmp()` with a pointer to this structure, you can return to this place in the code at any time until this function returns by calling the `longjmp()` primitive. The first argument is a pointer to the same structure, and the second argument may be any nonzero value.

The first time `setjmp()` is called, it returns a zero value. Each time `longjmp()` is called, Epsilon acts as if it is returning from the original `setjmp()` call again, returning the second argument from the `longjmp()`. For example:

```
one()
{
jmp_buf location;

if (setjmp(&location)){
        stuff("Back in one\n");
        return;
} else
        stuff("Ready to go\n");
two(&location);
}

two(loc)
jmp_buf *loc;
{
stuff("In two\n");
longjmp(loc, 1);
stuff("Never get here\n");
}
```

This example inserts the lines

```
Ready to go
In two
Back in one
```

```
jmp_buf *top_level;
```

The `error()` primitive uses the jump buffer pointed to by the `top_level` variable. If you wish to get control when the user presses the abort key, temporarily change the value of `top_level` to refer to another jump buffer. Make sure you restore it, however, or subsequent aborting may not work.

8.5.2 Character Types

```
int isspace(int ch)
int isdigit(int ch)
int isalpha(int ch)
int islower(int ch)
int isupper(int ch)
int isalnum(int ch)  /* basic.e */
int isident(int ch)  /* basic.e */
int any_uppercase(char *p)
```

Epsilon has several primitives that are helpful for determining if a character is in a certain class. The `isspace()` primitive tells if its character argument is a space, tab, or newline character. It returns 1 if it is, otherwise 0.

In the same way, the `isdigit()` primitive tells if a character is a digit (one of the characters 0 through 9), and the `isalpha()` primitive tells if the character is a letter. The `islower()` and `isupper()` primitives tell if the character is a lower case letter or upper case letter, respectively.

The `isalnum()` subroutine returns nonzero if the specified character is alphanumeric: either a letter or a digit. The `isident()` subroutine returns nonzero if the specified character is an identifier character: a letter, a digit, or the _ character.

The `any_uppercase()` subroutine returns nonzero if there are any upper case characters in its string argument p.

```
int tolower(int ch)
int toupper(int ch)
```

The `tolower()` primitive converts an upper case letter to the corresponding lower case letter. It returns a character that is not an upper case letter unchanged. The `toupper()` primitive converts a lower case letter to its upper case equivalent, and leaves other characters unchanged.

```
int set_character_property(int ch, int propcode, int value)
```

You can alter the rules Epsilon uses for determining if a particular character is alphabetic, uppercase, or lowercase, and how Epsilon case-folds when searching, sorting or otherwise comparing text, using the `set_character_property()` primitive. It takes the numeric code of the character whose properties you want to modify, a property code indicating which of its properties to access, and a new value for that property.

The property code `CPROP_CTYPE` sets whether the `isalpha()`, `isupper()`, `islower()`, and `isdigit()` primitives consider a character alphabetic, uppercase, lowercase, or a digit, respectively. These attributes are independent, though there are conventions for their use. (For instance, only alpha characters generally have a case, no character is both uppercase and lowercase, and so forth.) The bits `C_ALPHA`, `C_LOWER`, `C_UPPER`, and `C_DIGIT` represent these attributes. The bits also control whether the regular expressions `<digit>`, `<alpha>`, `<alphanum>`, and `<word>` match these characters; see page 58.

The property code `CPROP_TOLOWER` controls what value the `tolower()` primitive returns for the specified character, and the property code `CPROP_TOUPPER` controls what value the `toupper()` primitive returns for it.

The property code `CPROP_FOLD` controls how Epsilon case-folds that character during searching, sorting, and similar functions, whenever case folding is in use. It specifies a replacement character to be used in place of the original during comparisons. The complete set of case-folding properties must follow two rules: if some character X folds to Y, then Y must fold to itself, and character codes below 256 must never fold to a value greater than or equal to 256. (If a particular group of characters should be treated as equal when searching, setting the case folding property of each to the code of the lowest-numbered one is sufficient to comply with these rules.)

The primitive returns the previous value of the specified property of that character. If the new value is out of range for the property (such as a negative value), it will be ignored, and the primitive will just return the current value. You can use this to retrieve the current properties of a character without changing them.

Epsilon doesn't store current character properties in its state file. If you want to use non-default properties all the time, write a startup function that calls this primitive. See page 299.

Epsilon always starts with character classifications based on standard Unicode properties, except for the Win32 console version. That version, when running with a DOS/OEM character set (see the `console-ansi-font` variable), begins with its classifications for 8-bit characters set to match the current OEM font.

```
int get_direction()          /* window.e */
```

The `get_direction()` subroutine converts the last key pressed into a direction. It understands arrow keys, as well as the equivalent control characters. It returns BTOP, BBOTTOM, BLEFT, BRIGHT, or −1 if the key doesn't correspond to any direction.

8.5.3 Strings

```
int strlen(char *s)
```

Epsilon provides various primitives for manipulating strings, or equivalently, zero-terminated arrays of characters. The `strlen()` primitive returns the length of a string. That is, it tells the position in the array of the first zero character.

```
strcpy(char *tostr, char *fromstr)
strncpy(char *tostr, char *fromstr, int count)
```

The `strcpy()` primitive copies the null-terminated string `fromstr` to the array at `tostr`, including the terminating null character. The `strncpy()` primitive does the same, but always stops when `count` characters have been transferred, adding an additional null character to the string at `tostr` if necessary.

```
strcat(char *tostr, char *fromstr)
strncat(char *tostr, char *fromstr, int count)
```

The `strcat()` primitive concatenates (or appends) the string at `fromstr` after the string at `tostr`. For example, if `fromstr` points at the constant string "def" and `tostr` is an array of 10 characters that contains "abc" (and then, of course, a null character, plus 6 more characters with any value), then `strcat(tostr, fromstr);` makes the array `tostr` contain "abcdef" followed by a null character and 3 unused characters.

The `strncat()` primitive works similarly. It appends at most `count` characters from `fromstr`, and ensures that the result is zero-terminated by adding a null character if necessary. Note that the count limits the number of characters appended, not the total number of characters in the string.

```
int strcmp(char *first, char *second)
int strncmp(char *first, char *second, int count)
```

The `strcmp()` primitive tells if two strings are identical. It returns 0 if all characters in them are the same (and if they have the same length). Otherwise, it returns a negative number if the lexicographic ordering of these strings would put the first before the second. It returns a positive number otherwise. The `strncmp()` primitive is like `strcmp()`, except only the first `count` characters matter.

```
int strfcmp(char *first, char *second)
int strnfcmp(char *first, char *second, int count)
int charfcmp(int first, int second)
```

Epsilon also has similar comparison primitives that consider upper case and lower case letters to be equal. The `strfcmp()` primitive acts like `strcmp()` and the `strnfcmp()` primitive acts like `strncmp()`, but if the buffer-specific variable `case_fold` is nonzero, Epsilon folds characters in the same way searching or sorting would before making the comparison. The `charfcmp()` primitive takes two characters and performs the same comparison on them. For characters *a* and *b*, `charfcmp('`*a*`', '`*b*`')` equals `strfcmp("`*a*`", "`*b*`")`. (EEL also recognizes the corresponding ANSI C name `stricmp()` instead of `strfcmp()`.)

```
int compare_chars(char *str1, char *str2, int num, int fold)
set_chars(char *ptr, char value, int count)
```

The `compare_chars()` primitive works like `strcmp()`, except that it makes no assumptions about zero-termination. It takes two strings and a size, then compares that many characters from each string. If the strings exactly match, `compare_chars()` returns zero. If `str1` would be alphabetically before `str2`, it returns a negative value. If `str2` would be alphabetically before `str1`, it returns a positive value. It ignores the case of the characters when comparing if `fold` is nonzero.

The `set_chars()` primitive sets all the `count` characters in a character array `ptr` to the given `value`.

```
char *index(char *s, int ch)
char *rindex(char *s, int ch)
char *strstr(char *s, char *t)
```

The `index()` primitive tells if a character `ch` appears in the string `s`. It returns a pointer to the first appearance of `ch`, or a null pointer if there is none. The `rindex()` primitive works the same, but returns a pointer to the last appearance of `ch`. (EEL also recognizes the corresponding ANSI C names `strchr()` instead of `index()` and `strrchr()` instead of `rindex()`.)

The `strstr()` primitive searches the string `s` for a copy of the string `t`. It returns a pointer to the first appearance of `t`, or a null pointer if there is none. It case-folds as described above for `strfcmp()`.

```
int fpatmatch(char *s, char *pat, int prefix, int fold)
```

The `fpatmatch()` primitive returns nonzero if a string `s` matches a pattern `pat`. It uses a simple filename-style pattern syntax: * matches any number of characters; ? matches a single character, and [a-z] match a character class (with the same character class syntax as other patterns in Epsilon). If `prefix` is nonzero, `s` must begin with text matching `pat`; otherwise `pat` must match all of `s`. If `fold` is nonzero, Epsilon folds characters before comparing according to the current buffer's folding rules.

```
int sprintf(char *dest, char *format, ...)
```

The `sprintf()` primitive is the most powerful string building primitive Epsilon provides. It takes two or more arguments. The first is a character array. The remaining arguments are in the format that `say()` uses: a format string possibly followed by more arguments. (See page 235.) Instead of printing the string that is built on the screen, it copies the string into the destination array, and returns the number of characters copied.

```
int expand_string_template(char *dest, char *template,
                           char *keys, char *vals[])
```

The `expand_string_template()` subroutine uses a template to construct a new string `dest`. The template contains zero or more escape sequences, each a percent character % followed by another letter. The subroutine replaces each escape sequence with its corresponding value. The allowed escape sequence characters should be listed in `keys`, and their replacement values should be listed in the array of strings `vals` in the same order. Epsilon will interpolate the sequences to construct `dest`, returning nonzero to indicate an error in the template (usually an unknown escape sequence).

For instance, if `keys` contains `"ad"`, and the supplied `vals` array has been set so `vals[0]` is `"happy"` and `vals[1]` is `"tiger"`, then the template `"the %a is %d today"` is copied to `dest` as `"the tiger is happy today"`, and the subroutine will return 0.

8.5.4   Byte Arrays

These functions operate on 8-bit byte arrays, not 16-bit characters.

```
int memcmp(byte *str1, byte *str2, int num)
int memfcmp(byte *str1, byte *str2, int num)

memcpy(byte *tostr, byte *fromstr, int num)
memset(byte *ptr, char value, int count)
```

The `memcmp()` and `memfcmp()` primitives compare 8-bit bytes, not 16-bit characters like the `compare_chars()` and `strcmp()` primitives do. The `memfcmp()` primitive ignores case, `memcmp()` doesn't. They return values like the others, and don't stop comparing when an array element has a zero value, as `strcmp()` does.

The `memcpy()` primitive copies exactly `num` bytes from the second byte array to the first.

The `memset()` primitive sets all the `count` bytes in a byte array `ptr` to the given `value`.

```
chars_to_bytes(byte *b, char *s)
bytes_to_chars(char *s, byte *b)
```

The `chars_to_bytes()` function copies the null-terminated character string *s* to the byte array *b*, discarding the upper 8 bits of each 16-bit character.

The `bytes_to_chars()` function copies the null-terminated byte string *b* to the character array *s*.

8.5.5 Memory Allocation

```
char *malloc(int size)
char *realloc(char *ptr, int size)
free(char *ptr)
```

Epsilon maintains a pool of memory and provides primitives for allocating and deallocating blocks of any size. The `malloc()` primitive takes an int giving the number of characters of space required, and returns a pointer to a block of that size.

The `realloc()` primitive takes a pointer previously allocated with `malloc()`. First, it tries to expand the block to the requested size. If it cannot do that, it allocates another block of the requested size, then copies the old characters to the new block. In either case, it returns a pointer to a block of the requested size.

The `free()` primitive takes a pointer that `malloc()` previously returned and puts it back into the storage pool. Never use a block after you free it.

```
char *strsave(char *s)
```

For convenience, Epsilon provides a primitive to copy a string to an allocated block of the proper size. The `strsave()` primitive is used when a string needed later is stored in an array that must be reused. The primitive returns a pointer to the copy of the string it makes. The `free()` primitive may be given this pointer when the string is no longer needed.

```
user int availmem;
user int mem_in_use;
```

The `availmem` variable indicates the amount of free memory in the system, in environments where Epsilon knows this. The `mem_in_use` variable gives the space in bytes Epsilon is now using for miscellaneous storage (not including buffer text).

```
set_swapname(char *path)
```

If Epsilon can't fit all your files in available memory, it will swap parts to disk. The parts are contained in one or more swap files. The `set_swapname()` primitive tells Epsilon what directories to use for swap files, if it needs them. The argument is a string containing a list of *directories* in which to place swap files, as described under the `-fs` command line flag. After swapping has begun, this primitive has no effect. Supplying an empty argument `""` makes Epsilon use the standard place for swapping, as described under the `-fs` command line switch on page 12.

8.5.6 The Name Table

```
int final_index()
```

Epsilon keeps track of all EEL variables, commands, subroutines, key tables, color schemes, and keyboard macros in its *name table*. Each of these items has an entry there that lists its name, type, value, and additional information. An EEL program can access the table using a numeric index, like an array index. The first valid index to the name table is 1, and the `final_index()` primitive returns the last valid index. The index is based on the order in which the names were defined.

All variables appear in the name table, including primitive variables. Primitive functions (like most of those defined in this chapter) and EEL's `#define` textual macros are not in the name table. A state file contains an exact copy of a name table (plus some additional information).

Each entry contains the name of the item, a type code, a debugging flag, a help file offset, and whatever information Epsilon needs internally to make use of the item. When executing an EEL program, Epsilon automatically uses the table to find the value of a variable, for example, or execute a command. You can manipulate the table with EEL functions.

```
int find_index(char *name)
```

There are two ways to get an index if you have the name of an item. The `find_index()` primitive takes an item name as a string and returns the index of that item, or `0` if there is no such item. If the item is an EEL command or subroutine, casting its function pointer to a short also yields the index. For example, `(short)` `forward_word` gives the index of the command forward-word if `forward_word()` has been declared previously in the source file the expression appears in.

```
char *name_name(int index)
int name_type(int index)        /* codes: */
#define NT_COMMAND     1        /* normal bytecode function */
#define NT_SUBR        2        /* hidden bytecode function */
#define NT_MACRO       3        /* keyboard macro */
#define NT_TABLE       4        /* key table */
#define NT_VAR         5        /* normal variable */
#define NT_BUFVAR      6        /* buffer-specific variable */
#define NT_WINVAR      7        /* window-specific variable */
#define NT_COLSCHEME   8        /* color scheme */
#define NT_AUTOLOAD    10       /* load cmd from file */
#define NT_AUTOSUBR    11       /* load subr from file */
```

The primitives `name_name()` and `name_type()` return the name and type of a table entry, respectively. They each take an index into the name table and return the desired information. The value returned by `name_name()` is only valid until the next call to this function. Copy the name if you want to preserve it.

The codes for `name_type()` are in the standard include file codes.h.

```
int try_calling(char *name)
```

The `try_calling()` primitive calls a subroutine or command if it exists and doesn't complain if the function does not exist. It takes the name of the function to call. It returns `0` if the function doesn't exist. The function it calls must not require arguments.

```
int call_with_arg_list(int func, char *argtypes, int intargs[],
                       char *strargs[], ?char **res)
```

The `call_with_arg_list()` primitive can call an EEL function that takes only integer and string parameters, specified by its name table index `func`. The caller must specify the types of the parameters to pass in `argtypes`, and supply the integer and string parameters in the arrays `intargs` and `strargs`, respectively. The `argtypes` value is a list of characters: "i" for an integer, or "s" for a string. For each "i", Epsilon uses the next entry in the `intargs` array, and for each "s", the `strargs` array.

If the EEL function returns an integer, the `call_with_arg_list()` primitive returns it. If it returns a character pointer, pass the address of a character pointer, and the primitive will fill it in with the return value.

The `func` parameter may refer to a keyboard macro if `argtypes` is empty.

```
int drop_name(char *name)
```

To delete an item from the name table, use the `drop_name()` primitive. It returns 0 if it deleted the name, 1 if there was no such name in the name table, and 2 if there was such a name but it couldn't be deleted because it is currently in use.

```
int replace_name(char *old, char *new)
```

The `replace_name()` primitive renames an item in the name table. It returns 0 if the name change was successful, 1 if the original name did not exist, and 2 if the name change was unsuccessful because another item had the new name already. Any references to the original item result in an error, unless you provide a new definition for it later.

Sometimes when writing an Epsilon extension, you may wish to redefine one of Epsilon's built-in subroutines (`getkey()`, for example) to do something in addition to its usual action. You can, of course, simply modify the definition of the function, adding whatever you want. Unfortunately, if someone else gives you an extension that modifies the same function, it will overwrite your version. You'll have the same problem when you get a new version of Epsilon—you'll have to merge your change by hand.

```
#define REPLACE_FUNC(ext, func)  ....
        /* definition omitted */
```

Alternatively, you can create an extension that modifies the existing version of a function, even if it's already been modified. The trick is to replace it with a function that calls the original function. This can be done from a `when_loading()` function by using the `replace_name()` and `drop_name()` primitives, but eel.h defines a macro that does all of this. The `REPLACE_FUNC()` macro takes the name of the extension you're writing, and the name of the existing subroutine you want to replace. It doesn't really matter what the extension name is, just so long as no other extension uses it.

Here's an example. Suppose you're writing an extension that displays "Hello, world" whenever you start Epsilon. You've decided to name the extension "hello", and you want Epsilon's `start_up()` function to do the work. Here's what you do:

```
new_hello_start_up()    /* will be renamed to start_up */
{
   say("Hello, world");
   hello_start_up(); /* call old (which will have this name) */
}

REPLACE_FUNC("hello", "start-up")
```

Notice the steps: first you have to define a function with a name of the form new_<extension-name>_<replaced-function-name>. Make sure it calls a function named <extension-name>_<replaced-function-name>. Then do the `REPLACE_FUNC()`, providing the two names. This will rename the current <replaced-function-name> to <extension-name>_<replaced-function-name>, then rename your function to <replaced-function-name>.

8.5.7 Built-in and User Variables

Variables that are automatically defined by Epsilon, and have no definition in eel.h, are called built-in variables. These include `point`, `bufnum`, and most of the primitive variables described in this chapter. All

such built-in variables have entries in Epsilon's name table, so that you can see and set them using commands like set-variable or set-any-variable. Built-in variables have a `name_type()` code of `NT_BUILTVAR`.

```
int get_num_var(int i)
set_num_var(int i, int value)

char *get_str_var(int i)
set_str_var(int i, char *value)
```

Epsilon has several primitives that let you get and set the value of numeric and string global variables (including both built-in and ordinary, user-defined variables). Each primitive takes a name table index `i`. The `get_num_var()` and `get_str_var()` primitives return the numeric or string value (respectively) of the indicated variable, while the `set_num_var()` and `set_str_var()` primitives set the variable. If you provide an index that doesn't refer to a variable of the correct type, the setting functions do nothing, while the getting functions return zero. (See the `vartype()` primitive below.) The string functions only operate on variables with a character pointer data type, not on character arrays. Use `varptr()` below to modify character arrays.

The set-variable command and similar functions look for and try to call a function named `when_setting_`*varname*`()` after setting a variable named *varname*. For most variables a function with that name doesn't exist, and nothing happens. The `want-code-coloring` variable is an example of a variable with a `when_setting()` function. Its `when_setting()` function sets various other variables to match `want-code-coloring`'s new value.

Any user attempts to set a variable (such as running set-variable or loading a command file) will call such a function, but an ordinary assignment statement in an EEL function will not. If you write an EEL function that sets a variable with a `when_setting()` function, you should call the function explicitly after setting the variable.

```
int name_user(int i)
set_name_user(int i, int is_user)
```

For each global variable, built-in or not, Epsilon records whether or not it is a "user" variable. Some commands such as set-variable only show user variables. Otherwise, Epsilon treats user variables the same as others. The `name_user()` primitive returns non-zero if the variable with the given name table index is a user variable, and the `set_name_user()` primitive sets whether a variable with a particular name table index is a user variable.

```
user int my_var;          // sample declaration
```

By default, variables you declare with EEL are all non-user variables, hidden from the user. If the user is supposed to set a variable directly in order to alter a command's behavior, put the `user` keyword before its global variable definition to make it a user variable. (In previous versions, Epsilon used a convention that any non-user variables you defined had to start with an underscore character, and all others were effectively user variables. This convention still works: set-variable will still exclude such variables from normal completion lists.)

```
int ptrlen(char *p, ?int in_bytes)
```

The `ptrlen()` primitive takes a pointer of any type and returns the size in characters of the object it points to. The value of `ptrlen(p)` is the lowest value `i` for which `((char *)p)[i]` is an illegal dereference. If its optional second argument is nonzero, it returns its count in bytes, not characters. (Characters are 16 bits wide, while bytes are 8 bits wide.)

```
char *varptr(int i)
int pointer_to_index(void *)
```

The `varptr()` primitive returns a pointer to any global variable given its index in the name table. The pointer is always a character pointer and should be cast to the correct type before it's used. When `varptr()` is applied to a buffer-specific or window-specific variable, Epsilon checks the `use_default` variable to determine if a pointer to the default or current value should be returned (see page 296). This function doesn't operate with built-in variables—use `get_num_var()` and similar functions for these.

The `pointer_to_index()` primitive does the reverse. It takes a pointer and checks to see if it refers to a global variable. If a global variable is an array or structure, the pointer can point anywhere within. It returns the name table index of the global variable, or `0` if the pointer doesn't point to the contents of any global variable.

```
int vartype(int i)

#define TYPE_CHAR      1
#define TYPE_SHORT     2
#define TYPE_INT       3
#define TYPE_CARRAY    4        /* character array */
#define TYPE_CPTR      5        /* character pointer */
#define TYPE_POINTER   6        /* contains pointers or spots */
#define TYPE_OTHER     7        /* none of the above */
```

The `vartype()` primitive returns information on the type of a global variable (or buffer-specific or window-specific variable). It takes the index of the variable in the name table and returns one of the above codes if the variable has type character, short, integer, character array, or character pointer. It returns `TYPE_POINTER` if the variable is a spot or pointer, or a structure or union containing a spot or pointer. Otherwise, it returns `TYPE_OTHER`.

```
int new_variable(char *name, int type, int vtype, ?int length)
```

The `new_variable()` primitive provides a way to create a new variable without having to load a bytecode file. The first argument specifies the name of the variable. The second argument is a type code of the kind returned by the `name_type()` primitive. The code must be `NT_VAR` for a normal variable, `NT_BUFVAR` for a buffer-specific variable, `NT_WINVAR` for a window-specific variable, or `NT_COLSCHEME` for a color scheme. The third argument is a type code of the kind returned by the `vartype()` primitive. This code must be one of the following: `TYPE_CHAR`, `TYPE_SHORT`, `TYPE_INT`, or `TYPE_CARRAY`. The last argument is a size, which is used only for `TYPE_CARRAY`. It returns the name table index of the new variable, or `-1` if it couldn't create the variable in question.

8.5.8   Buffer-specific and Window-specific Variables

```
char use_default;
```

Epsilon's buffer-specific variables have a value for each buffer. They change when the current buffer changes. When you create a new buffer, you also automatically create a new copy of each buffer-specific variable. The initial value of each newly created buffer-specific variable is set from special default values Epsilon maintains. These values may be set using the variable `use_default`. When `use_default` is nonzero, referencing any buffer-specific variable accesses its default value, not the value for the current buffer. Otherwise, a value particular to the current buffer applies, as usual.

The normal way to reference a variable's default value is to use the ".default" syntax described on page 161, not to set `use_default`.

Window-specific variables have a separate value for each window. When you split a window, the newly created window initially has the same values for all variables as the original window. Each window-specific variable also has a default value, which can be referred to in the same way as buffer-specific variables, via the ".default" syntax described on page 161 or by setting the `use_default` variable. Epsilon uses the default value to initialize the first window it creates, during startup, and when it creates pop-up windows.

Only the default values of window- and buffer-specific variables are saved in a state file.

```
copy_buffer_variables(int tobuf, int frombuf)
safe_copy_buffer_variables(int tobuf, int frombuf)
```

The `copy_buffer_variables()` primitive sets all buffer-specific variables in the buffer `tobuf` to their values in the buffer `frombuf`. If `frombuf` is zero, Epsilon resets all buffer-specific variables in the buffer `tobuf` to their default values. The `safe_copy_buffer_variables()` subroutine calls `copy_buffer_variables()`, then clears the values of certain variables that should not be copied between buffers; generally these variables are spot variables that must always refer to positions within their own buffers.

8.5.9 Bytecode Files

```
load_commands(char *file)
load_from_path(char *file)     /* control.e */
int load_eel_from_path(char *file, int gripe)
```

The `load_commands()` primitive loads a bytecode file of command, subroutine and variable definitions into Epsilon after the EEL compiler has produced it from the .e source file. The primitive changes the name provided so that it has the appropriate .b extension, then opens and reads the file. The primitive prints a message and aborts to top-level if it cannot find the file or the file name is invalid.

The subroutine `load_from_path()` searches for a bytecode file using the `lookpath()` primitive (see page 266) and loads it using `load_commands()`.

The `load_eel_from_path()` subroutine searches for an EEL source file with the specified name using `lookpath()`. Then it compiles and loads the file. The `gripe` parameter tells it when to report errors to the user. Zero means never report errors, one means complain if the file contained errors, and two means also complain if no file by that name was found. The subroutine returns zero if the file compiled and loaded without errors, one if it wasn't found, and two for other errors.

```
int eel_compile(char *file, int use_fsys, char *flags,
                char *errors, int just_check)
```

The `eel_compile()` primitive lets Epsilon run the EEL compiler without having to invoke a command processor. File specifies the name of a file or buffer. If `use_fsys` is nonzero, it names a file; if `use_fsys`

is zero, a buffer. The `flags` parameter may contain any desired command line flags. Compiler messages will go to the buffer named `errors`. Unless errors occur or the `just_check` parameter is nonzero, Epsilon will automatically load the result of the compilation. No bytecode file on disk will be modified. Note that when the compiler includes header files, it will always read them from disk, even if they happen to be in an Epsilon buffer.

The primitive returns `0` on success, `1` if the compilation had fatal errors and did not complete, `2` if the compiler could not be located, or `-1` if the user aborted.

```
when_loading()            /* EEL subroutine */
```

Any subroutines with the special name `when_loading()` execute as they are read, and then go away. There may be more than one of these functions defined in a single file. (Note: When the last function defined in an EEL file has been deleted or replaced, Epsilon discards all the constant strings defined in that file. So a file that contains only a `when_loading()` function will lose its constant strings as soon as it exits. If a pointer to such a string must be put in a global variable, use the `strsave()` primitive to make a copy of it. See page 291.)

The `autoload_commands()` primitive described below executes any `when_loading()` functions defined in the file, just as `load_commands()` would. Epsilon never arranges for a `when_loading()` function to be autoloaded, and will execute and discard such functions as soon as they're loaded. If you run `autoload_commands()` on a file with `when_loading()` functions, Epsilon will execute them twice: once when it initially sets up the autoloading, and once when it autoloads the file.

```
user char *byte_extension;
user char *state_extension;
```

The extensions used for Epsilon's bytecode files and state files may vary with the operating system. Currently, all operating system versions of Epsilon use ".b" for bytecode files, and ".sta" for state files. The `byte_extension` and `state_extension` primitives hold the appropriate extension names for the particular version of Epsilon.

```
autoload(char *name, char *file, int issubr)
autoload_commands(char *file)
```

Epsilon has a facility to define functions that are not loaded into memory until they are invoked. The `autoload()` primitive takes the name of a function to define, and the name of a bytecode file it can be found in. The file name string may be in a temporary area, because Epsilon makes a copy of it.

The primitive's final parameter should be nonzero to indicate that the autoloaded function will be a subroutine, or zero if the function will be a command. (Recall that commands are designed to be invoked directly by the user, and may not take parameters, while subroutines are generally invoked by commands or other subroutines, and may take parameters.) Epsilon enters the command or subroutine in its name table with a special code to indicate that the function is an autoloaded function: `NT_AUTOLOAD` for commands, or `NT_AUTOSUBR` for subroutines.

When Epsilon wants to call an autoloaded function, it first invokes the EEL subroutine `load_from_path()`, passing it the file name from the `autoload()` call. The standard definition of this function is in the file control.e. It searches for the file along the EPSPATH, as described in the manual, and then loads the file. The `load_from_path()` subroutine reports an error and aborts the calling function if it cannot find the file.

When `load_from_path()` returns, Epsilon checks to see if the function is now defined as a regular, non-autoloaded function. If it is, Epsilon calls it. However, it is not necessarily an error if the function is still

undefined. Sometimes a function's work can be done entirely by the `when_loading()` subroutines that are run and immediately discarded as a bytecode file loads.

For example, all the work of the set-color command was once done by a `when_loading()` function in the EEL file color.e. (In recent versions, it no longer uses autoloading.) Loading the corresponding bytecode file automatically ran this `when_loading()` function, which displayed some windows and let the user choose colors. When the user exited from the command, Epsilon discarded the code for the `when_loading()` function that displayed windows and interpreted keys, and finishes loading the bytecode file. The set-color command was still defined as a command that autoloads the color.b bytecode file, so the next time the user ran this command, Epsilon loaded the file again.

If the autoloaded function was called with parameters, but remains undefined after Epsilon tries to autoload it, Epsilon aborts the calling function with an error message. Functions that use the above technique to load temporarily may not take parameters.

Like `load_commands()`, the primitive `autoload_commands()` takes the name of a compiled EEL bytecode file as a parameter. It loads any variables or bindings contained in the file, just like `load_commands()`. But instead of loading the functions in the file, this primitive generates an autoload request for each function in the file. Whenever any EEL function tries to call a function in the file, Epsilon will load the entire file.

8.5.10   Starting and Finishing

```
do_save_state(char *file)
int save_state(char *file)
```

The `do_save_state()` subroutine writes the current state to the specified file. It aborts with an error message if it encounters a problem. It uses the `save_state()` primitive to actually write the state. The primitive returns 0 if the information was written successfully, or an error code if there was a problem (as with `file_write()`). Both change the extension to ".sta" before using the supplied name.

The state includes all commands, subroutines, keyboard macros, and variables. It does not include buffers or windows. Since a state file can only be read while Epsilon is starting (when there are no buffers or windows), only the default value of each buffer-specific or window-specific variable is saved in a state file.

Pointer variables will have a value of zero when the state file is loaded again. Epsilon does not save the object that is pointed to. Spot variables and structures or unions containing pointers or spots are also zeroed, but other types of variables are retrieved unchanged (but see the description of the `zeroed` keyword on page 182).

```
short argc;
char *argv[ ];
```

When Epsilon starts, it examines the arguments on its command line, and modifies its behavior if it recognizes certain special flags. (Before examining its command line, it types in the contents of the configuration variable EPSILON if this exists.) First, if necessary, it breaks the text of the command line up into individual words, separated by spaces. (Words enclosed in `"` characters may contain spaces.) It looks for certain special flags, interprets them and removes them from the command line. It then passes the remainder of the command line to the EEL startup code in cmdline.e. That code interprets any remaining flags and files on the command line. You can add new flags to Epsilon by modifying cmdline.e. See page 11 for the meaning of each of Epsilon's flags.

Epsilon interprets and removes these flags from the command line:

-k	Keyboard options	-m	Memory control
-s	Load from state file	-w	Directory options
-b	Load from bytecode file	-v	Video options

Some of these settings are visible to an EEL program through variables. See the `load-from-state` variable for the -b flag, the `state_file` variable for the -s flag, the `want-cols` and `want-lines` variables for the -vc and -vl flags, and the `directory-flags` variable for the -w flag.

All other flags, as well as any specified files, are interpreted by the EEL functions in cmdline.e. They read the command line from the `argc` and `argv` variables, already broken down into words. The `argc` variable contains the number of words in the command line. The `argv` variable contains the words themselves. The first word on the command line, `argv[0]`, is always the name of Epsilon's executable file, so that if `argc` is 2, there was one argument and it is in `argv[1]`.

```
when_restoring()        /* cmdline.e */
early_init()            /* cmdline.e */
middle_init()           /* cmdline.e */
start_up()              /* cmdline.e */
user char *version;
apply_defaults()
```

Epsilon calls the EEL subroutine `when_restoring()` if it exists after loading a state file. Unlike `when_loading()`, this subroutine is not removed after it executes. The standard version of `when_restoring()` sets up variables and modes, and interprets the command line. It calls several EEL subroutines at various points in the process. Each does nothing by default, but you can conveniently customize Epsilon by redefining them. (See page 293 to make sure your extension doesn't interfere with other extensions.)

The `when_restoring()` function calls `early_init()` just before interpreting flags, and `middle_init()` just after. It then loads files (from the command line, or a saved session), displays Epsilon's version number, and calls the `start_up()` subroutine. (The `version` variable contains a string with the current version of Epsilon, such as "9.0".) Finally, Epsilon executes any -l and -r switches.

The `when_restoring()` subroutine calls the `apply_defaults()` primitive before it calls `early_init()`. This primitive sets the values of window-specific and buffer-specific variables in the current buffer and window to their default values.

```
char state_file[ ];
user char load_from_state;
```

The `state_file` primitive contains the name of the state file Epsilon was loaded from, or `""` if it was loaded only using bytecode files with the -b flag. The `load_from_state` variable will be set to 1 if Epsilon loaded its functions from a state file at startup, or 0 if it loaded only from bytecode files.

```
after_loading()
```

After Epsilon calls the `when_restoring()` subroutine, it finishes its internal initialization by checking for the existence of certain variables and functions that must be defined if Epsilon is to run. Until this is done, Epsilon can't perform a variety of operations such as getting a key from the keyboard, displaying buffers, and searching. The `after_loading()` primitive tells Epsilon to finish initializing now. The variables and functions listed in figure 8.2 must be defined when you call `after_loading()`.

```
when_idle()
when_displaying()
when_repeating()
getkey()
on_modify()
prepare_windows()
build_mode()
fix_cursor()
load_from_path()
color_class standard_color;
color_class standard_mono;
user int see_delay;
user short beep_duration;
user short beep_frequency;
user char mention_delay;
char _display_characters[ ];
user buffer int undo_size;
buffer short *mode_keys;
user buffer short tab_size;
user buffer short case_fold;
buffer char *_display_class;
char *_echo_display_class;
user window int display_column;
window char _highlight_control;
window char _window_flags;
char use_process_current_directory;
```

Figure 8.2: Variables and functions that must be defined.

```
finish_up()
user char leave_blank;
```

When Epsilon is about to exit, it calls the subroutine `finish_up()`, if it exists. (See page 293 to make sure your extension doesn't interfere with other extensions that may also define `finish_up()`.) Epsilon normally redisplays each mode line one last time just before exiting, so any buffers that it saved just before exiting will not still be marked unsaved on the screen. However, if the `leave_blank` primitive is nonzero, it skips this step.

8.5.11   EEL Debugging and Profiling

```
int name_debug(int index)
set_name_debug(int index, int flag)
```

Every command or subroutine in Epsilon's name table has an associated debug flag. If the debug flag of a command or subroutine is nonzero, Epsilon will start up the EEL debugger when the function is called, allowing you to step through the function line by line. See page 136. The `name_debug()` primitive returns the debug flag for an item, and the `set_name_debug()` primitive sets it.

```
start_profiling()
stop_profiling()
char *get_profile()
```

Epsilon can generate an execution profile of a section of EEL code. A profile is a tool to determine which parts of a program are taking the most time. The `start_profiling()` primitive begins storing profiling information internally. Profiling continues until Epsilon runs out of space, or you call the `stop_profiling()` primitive, which stops storing the information. Many times each second, Epsilon saves away information describing the location in the source file of the EEL code it is executing, if you've turned profiling on. You can use this to see where a command is spending its time, so that you can center your efforts to speed the command up there.

Once you stop the profiling with the `stop_profiling()` primitive, you can retrieve the profiling information with the `get_profile()` primitive. Each call returns one line of the stored profile information, and the function returns a null pointer when all the information has been retrieved. Each line contains the name of an EEL source file and a line number within the file, separated by a space. See the profile command for a more convenient way to use these primitives. Functions that you've compiled with the EEL compiler's `-s` flag will not appear in the profile.

8.5.12   Help Subroutines

```
int name_help(int index)
set_name_help(int index, int offset)
get_doc()                    /* help.e */
```

Every item in Epsilon's name table has an associated help file offset. The help offset contains the position in Epsilon's help file "edoc" where information on an item is stored. Epsilon uses it to provide quick access to help file items. It is initially -1, and may be set with the `set_name_help()` primitive and examined with the `name_help()` primitive. (The Windows version of Epsilon normally uses a standard Windows help file to display help, so it doesn't use these help file offsets.)

When an EEL function wants to look up information in the help file, it calls the EEL subroutine `get_doc()`. This function loads the help file into the buffer "-edoc" if it hasn't been loaded before.

Epsilon's help file "edoc" uses a simple format that makes it easy to add new entries for your own commands. Each command's description begins with a line consisting of a tilde (˜), the command or variable's name, a ⟨Tab⟩, and the command's one-line description (or, for a variable, some type information). Following lines (until the next line that starts with ˜, or the end of the file) constitute the command's full description. The entries can occur in any order; they don't have to be listed alphabetically.

An entry can contain a cross-reference link to another entry in the file; these consist of the name of the command or variable being cross-referenced, bracketed by two control characters. Put a ˆA character before the name of the command or variable, and a ˆB character after. Also see the description of the `view_linked_buf()` subroutine on page 221.

```
help_on_command(int ind)            /* help.e */
help_on_current()                   /* help.e */
```

The `help_on_command()` subroutine provides help on a particular command. It takes the name table index of the command to provide help on.

The `help_on_current()` subroutine displays help on the currently-running command. It uses the `last_index` variable to determine the current command.

```
show_binding(char *fmt, char *cmd)  /* help.e */
```

The `show_binding()` subroutine displays the message `fmt` using the `say()` primitive. The `fmt` must contain the `%s` sequence (and no other `%` sequences). Epsilon will replace the `%s` with the binding of the command `cmd`. For example,

```
show_binding("Type %s to continue", "exit-level");
```

displays "Type Ctrl-X Ctrl-Z to continue" with Epsilon's normal bindings.

8.6   Input Primitives

8.6.1   Keys

```
wait_for_key()
user int key;
user int full_key;
int generic_key(int k)
when_idle(int times)      /* EEL subroutine */
add_buffer_when_idle(int buf, int (*func)())
delete_buffer_when_idle(int buf, int (*func)())
when_repeating()          /* EEL subroutine */
int is_key_repeating()
```

The `wait_for_key()` primitive advances to the next key, waiting for one if necessary. The variable `key` stores the last key obtained from `wait_for_key()`.

Some key combinations have both a generic and specific interpretation. For instance, the ⟨Plus⟩ key on a numeric keypad has a generic code identical to the ⟨Plus⟩ key on the main keyboard, but a unique specific

code. Epsilon sets the `full_key` variable to the specific key code for that key, and `key` to the generic code for the key (in this case, "+"). For keys with only one interpretation, Epsilon sets `key` and `full_key` the same.

The `generic_key()` primitive returns the generic version of the specified key. For keys with only one interpretation, it returns the original key code.

When you call `wait_for_key()`, it first checks to see if the `ungot_key` variable has a key (see below) and uses that if it does. If not, and a keyboard macro is active, `wait_for_key()` returns the next character from the macro. (The primitive also keeps track of repeat counts for macros.) If there is no key in `ungot_key` and no macro is active, the primitive checks to see if you have already typed another key and returns it if you have. If not, the primitive waits until you type a key (or a mouse action or other event occurs—Epsilon treats all of these as keys).

When a concurrent process is outputting text into an Epsilon buffer, it only appears there during a call to `wait_for_key()`. Epsilon handles the processing of other concurrent events like FTP transfers during this time as well.

While Epsilon is waiting for a key, it calls the `when_idle()` subroutine. The default version of this function does idle-time code coloring and displays any defined idle-time message in the echo area (see the `show-when-idle` variable), among other things. The `when_idle()` subroutine receives a parameter that indicates the number of times the subroutine has been called since Epsilon began waiting for a key. Every time Epsilon gets a key (or event), it resets this count to zero.

The `when_idle()` subroutine should return a timeout code in hundredths of a second. Epsilon will not call the subroutine again until the specified time has elapsed, or another key arrives. If it doesn't need Epsilon to call it for one second, for example, it can return `100`. If it wants Epsilon to call it again as soon as possible (assuming Epsilon remains idle), it can return `0`. If the subroutine has completed all its work and doesn't need to be called again until after the next keystroke or mouse event, it can return `-1`. Epsilon will then go idle waiting for the next event. (The return value is only advisory; Epsilon may call `when_idle()` more frequently or less frequently than it requests.)

A mode may wish to provide additional functions that run during idle time, beyond those the `when_idle()` subroutine performs itself. The `add_buffer_when_idle()` subroutine registers a function `func` so that it will be called during idle-time processing whenever `buf` is the current buffer. The `delete_buffer_when_idle()` subroutine removes the specified function from that buffer's list of buffer-specific idle-time functions. (It does nothing if the function was not on the list.) A buffer-specific when-idle function takes a parameter `times` and must return a result in the same fashion as the `when_idle()` function itself.

When you hold down a key to make it repeat, Epsilon does not call the `when_idle()` subroutine. Instead, it calls the `when_repeating()` subroutine. Again, this varies by environment: under some operating systems, Epsilon cannot distinguish between repeated key presses and holding down a key to make it repeat. If this is the case, Epsilon won't call the function.

The `is_key_repeating()` primitive returns nonzero if the user is currently holding down a key causing it to repeat. Epsilon can't detect this in all environments, so the primitive always returns `0` in that case.

```
int getkey()            /* control.e */
```

Instead of calling `wait_for_key()` directly, EEL commands should call the EEL subroutine `getkey()` (defined in control.e), to allow certain actions that are written in EEL code to take effect on each character. For example, the standard version of `getkey()` saves each new character in a macro, if you're defining one. It checks the EEL variable `_len_def_mac`, which contains the length of the macro being defined plus one, or zero if you're not defining a macro. For convenience, `getkey()` also returns the new key. The `getkey()`

subroutine calls `wait_for_key()`. (If you want to add functions to `getkey()`, see page 293 to make sure your extension doesn't interfere with other extensions that may also add to `getkey()`.)

```
int char_avail()
int in_macro(?int ignore_suspended)
```

The `char_avail()` primitive returns 0 if `wait_for_key()` would have to wait if it were called, and 1 otherwise. That is, it returns nonzero if and only if a key is available from `ungot_key`, a keyboard macro, or the keyboard.

The `in_macro()` primitive returns 1 if a keyboard macro is running or has been suspended, 0 otherwise. If its optional `ignore_suspended` parameter is nonzero, it counts a suspended macro as if no macro were running. While processing the last key of a keyboard macro, `in_macro()` will return 0, because Epsilon has already discarded the keyboard macro by that time. Check the `key-from-macro` variable instead to see if the key currently being handled came from a macro.

There are some textual macros defined in eel.h which help in forming the codes for keys in an EEL function. The codes for normal ASCII keys are their ASCII codes, so the code for the key 'a' is `'a'`; the same goes for Unicode characters. The `ALT()` macro makes these normal keys into their Alt forms, so the code for Alt-a is `ALT('a')`. The `CTRL()` macro changes a character into the corresponding control character, so `CTRL('h')` or `CTRL('H')` both represent the Ctrl-h key. Both `CTRL(ALT('q'))` and `ALT(CTRL('q'))` stand for the Ctrl-A-q key.

The remaining key codes represent those keys that don't correspond to any possible buffer character, plus various key codes that represent other kinds of input events, such as mouse activity.

The `FKEY()` macro represents the function keys. `FKEY(1)` and `FKEY(12)` are F1 and F12, respectively. Note that this macro takes a number, not a character.

Refer to the cursor pad keys using the macros `KEYINSERT`, `KEYEND`, `KEYDOWN`, `KEYPGDN`, `KEYLEFT`, `KEYRIGHT`, `KEYHOME`, `KEYUP`, `KEYPGUP`, and `KEYDELETE`. You can refer to the numeric keypad keys with the `NUMDIGIT()` macro: `NUMDIGIT(0)` is N-0, and `NUMDIGIT(9)` is N-9. `NUMDOT` is the numeric keypad period, and `NUMENTER` is the ⟨Enter⟩ or ⟨Return⟩ key on the numeric keypad (normally mapped to Ctrl-M).

The codes for the grey keys are `GREYPLUS`, `GREYMINUS`, `GREYSTAR` and `GREYSLASH` for the +, −, *, and / keys on the numeric keypad, and `GREYENTER`, `GREYBACK`, `GREYTAB`, `GREYESC`, and `SPACEBAR` for the ⟨Enter⟩, ⟨Backspace⟩, ⟨Tab⟩, ⟨Esc⟩, and ⟨Spacebar⟩ keys, respectively.

```
#define NUMSHIFT(c)     ((c) | KEY_SHIFT)
#define NUMCTRL(c)      ((c) | KEY_CTRL)
#define NUMALT(c)       ((c) | KEY_ALT)
#define KEY_PLAIN(c)    ((c) & ~ (KEY_SHIFT | KEY_CTRL | KEY_ALT))
```

The `NUMSHIFT()`, `NUMCTRL()`, and `NUMALT()` macros make shifted, control, and alt versions of keys, respectively, by turning on the bit in a key code for each of these properties: `KEY_SHIFT`, `KEY_CTRL`, and `KEY_ALT`. For example, `NUMCTRL(NUMDIGIT(3))` is Ctrl-N-<PgDn>, and `NUMALT(KEYDELETE)` is A-<Del>. The `KEY_PLAIN()` macro strips away these bits.

In this version, `NUMALT()` and `ALT()` are the same, and `NUMCTRL()` and `CTRL()` only differ on certain low-numbered characters: the former always turns on the `KEY_CTRL` bit, while the latter also generates ASCII control codes when given suitable ASCII characters.

```
int make_alt(int k)          /* control.e */
int make_ctrl(int k)         /* control.e */
```

The `make_alt()` subroutine defined in control.e will return an Alt version of any key. The `make_ctrl()` subroutine is similar, but makes a key into its Control version. These may be used instead of the `ALT()` and `CTRL()` macros.

Use the `IS_CTRL_KEY()` macro to determine if a given key is a control key of some kind. Its value is nonzero if the key is an ASCII Control character, a function key with Control held down, or any other Control key. It understands all types of keys. The macro `IS_ALT_KEY()` is similar; its value is nonzero if the given key was generated when holding down the Alt key.

A macro command recorded using the notation `<!find-file>` uses the bit flag `CMD_INDEX_KEY`. In this case the value of `key` is not a true key, but rather the name table index of the specified command. See page 139 for more information.

```
user int ungot_key;
```

If the `ungot_key` variable is set to some value other than its usual value of -1, that number is placed in `key` and `full_key` as the new key when `wait_for_key()` is called next, and `ungot_key` is set to -1 again. You can use this to make a command that reads keys itself, then exits and runs the key again when you press an unrecognized key. The statement `ungot_key = key;` accomplishes this.

```
show_char(char *str, int key, ?int style)
```

The `show_char()` primitive converts a key code to its printed representation, described on page 139. For example, the code produced by function key 3 generates the string `F-3`. The string is *appended* to the character array `str`.

If `show_char()`'s optional third parameter is present, and nonzero, this primitive will use a longer, more readable printed representation. For example, rather than `C-A-S` or `,` or `S-F-10`, `show_char()` will return `Ctrl-Alt-S` or `<Comma>` or `Shift-F10`. (Epsilon can only parse the former style, in Epsilon command files and in all other commands that use the `get_keycode()` primitive below.)

See the `%k` sequence used by `sprintf()` and other functions, described on page 235, for a more convenient way to translate keys or characters to text.

```
#define key_t           int
key_t *get_keycode()
int key_value(char *s, ?char **after)
stuff_macro(key_t *mac, int oneline)
```

The `get_keycode()` primitive is used to translate a sequence of key names such as `"C-xC-A-f"` into the equivalent key codes. It moves past a quoted sequence of key names in the buffer and returns an array of ints with the key codes. The same array is used each time the function is called. The first entry of the array contains the number of array entries. The primitive returns null if the string had an invalid key name.

The `key_t` macro represents the type of a key. It's the same as an `int` in this versions, but not in older versions. Writing `key_t` instead of `int`, and `#including` the compatibility header file `oldkeys.h` helps make EEL code compatible with older versions.

The `key_value()` primitive also converts key names into key codes, but it gets the key name from a string, not a buffer, and returns a single key code at a time. It tries to interpret `s` as a key name, and returns its value. If the optional pointer `after` is non-null, it must point to a character pointer. Epsilon sets `*after` to the position in the string after the key name. When `s` contains an invalid key name, `key_value()` returns -1 and sets `*after` (if `after` is non-null) to `s`.

The `stuff_macro()` subroutine inserts a sequence of key names into the current buffer in a format that `get_keycode()` can read, surrounding the key names with `"` characters. The list of keys is specified by an array of ints in the same format `get_keycode()` uses: the first value contains the total number of array entries. If `oneline` is nonzero, the subroutine represents line breaks with `\n` so that the text stays on one line.

```
user char key_type;
user short key_code;
```

When `wait_for_key()` returns a key that comes directly from the keyboard, it also sets the primitive variables `key_type` and `key_code`. These let EEL programs distinguish between keys that translate to the same Epsilon key code, for certain special applications. The `wait_for_key()` primitive doesn't change either variable when the key comes from `ungot_key`.

The `key_code` variable contains the sixteen-bit BIOS-style encoding for the key that Epsilon received from the operating system, if available. Its ASCII code is in the low eight bits and its scan code is in the high eight bits.

The `key_type` variable has one of the following values, defined in codes.h. If `KT_NONASCII` or `KT_NONASCII_EXT`, the key was a special key without an ASCII translation, such as a function key. Such keys are of type `KT_NONASCII_EXT` if they're one of the keys on an extended keyboard that are synonyms to multikey sequences on the old keyboard, such as the keys on the extended keyboard's cursor pad.

A key type of `KT_ACCENT_SEQ` indicates a multikey sequence that the operating system or a resident program has translated as a single key, such as an ê. Key type `KT_ACCENT` generally means the operating system translated a single key to a graphics character or foreign language character. Key type `KT_NORMAL` represents any other key. Most keys have a key type of `KT_NORMAL`.

A key type of `KT_MACRO` means the key came from a macro. A macro key recorded with the `EXTEND_SEL_KEY` bit flag returns a key type of `KT_EXTEND_SEL` instead, but these extend codes are not used in current versions of Epsilon. In either case, the `key_code` variable is set to zero in this case.

In many environments, the `key_code` variable is always zero, and `key_type` is either `KT_NORMAL`, `KT_MACRO`, or `KT_EXTEND_SEL`.

8.6.2   The Mouse

When a mouse event occurs, such as a button press or a mouse movement, Epsilon enqueues the information in the same data structure it uses for keyboard events. A call to `wait_for_key()` retrieves the next item from the queue—either a keystroke or a mouse event. Normally an EEL program calls the `getkey()` subroutine instead of `wait_for_key()`.

```
user short catch_mouse;
```

The `catch_mouse` primitive controls whether Epsilon will queue up any mouse events. Setting it to zero causes Epsilon to ignore the mouse. A nonzero value makes Epsilon queue up mouse events. If your system has no mouse, setting `catch_mouse` has no effect.

```
user short mouse_mask;
user short mouse_x, mouse_y;
user short mouse_screen;
user int double_click_time;
```

You can control which mouse events Epsilon dequeues, and which it ignores, by using the `mouse_mask` primitive. The following values, defined in codes.h, control this:

```
#define MASK_MOVE       0x01
#define MASK_LEFT_DN    0x02
#define MASK_LEFT_UP    0x04
#define MASK_RIGHT_DN   0x08
#define MASK_RIGHT_UP   0x10
#define MASK_CENTER_DN  0x20
#define MASK_CENTER_UP  0x40
#define MASK_ALL        0x7f
#define MASK_BUTTONS    (MASK_ALL - MASK_MOVE)
#define MASK_DN         // ... see eel.h
#define MASK_UP         // ... see eel.h
```

For example, the following EEL code would cause Epsilon to pay attention to the left mouse button and mouse movement, but ignore everything else:

```
mouse_mask = MASK_MOVE | MASK_LEFT_DN | MASK_LEFT_UP;
```

When Epsilon dequeues a mouse event with `wait_for_key()`, it sets the values of `mouse_x` and `mouse_y` to the screen coordinates associated with that mouse event. Setting them moves the mouse cursor. The upper left corner has coordinate (0, 0).

When dequeuing a mouse event, `wait_for_key()` returns one of the following "keys" (defined in codes.h):

```
MOUSE_LEFT_DN      MOUSE_LEFT_UP      MOUSE_DBL_LEFT
MOUSE_CENTER_DN    MOUSE_CENTER_UP    MOUSE_DBL_CENTER
MOUSE_RIGHT_DN     MOUSE_RIGHT_UP     MOUSE_DBL_RIGHT
MOUSE_MOVE
```

Dequeuing a mouse event also sets the `mouse_screen` variable to indicate which screen its coordinates refer to. Screen coordinates are relative to the specified screen. Ordinary Epsilon windows are on the main screen, screen 0. When Epsilon creates a dialog box containing Epsilon windows, each Epsilon window receives its own screen number. For example, if you type `Ctrl-X Ctrl-F ?`, Epsilon displays a dialog box with two screens, usually numbered 1 and 2. If you click on the ninth line of the second screen, Epsilon returns the key MOUSE_LEFT_DN, sets `mouse_y` to 8 (counting from zero), and sets `mouse_screen` to 2.

The `double_click_time` primitive specifies how long a delay to allow for double-clicks (in hundredths of a second). If two consecutive MOUSE_LEFT_DN events occur within the allotted time, then Epsilon enqueues a MOUSE_DBL_LEFT event in place of the second MOUSE_LEFT_DN event. The corresponding thing happens for right clicks and center clicks as well. Epsilon for Windows ignores this variable and uses standard Windows settings to determine double-clicks.

```
#define IS_WIN_KEY(k)        // ... omitted
#define IS_MOUSE_KEY(k)      // ... omitted
#define IS_TRUE_KEY(k)       // ... omitted
#define IS_EXT_ASCII_KEY(k)  // ... omitted
#define IS_WIN_PASSIVE_KEY(k) // ... omitted
#define IS_MOUSE_LEFT(k)     // ... omitted
```

```
    #define IS_MOUSE_RIGHT(k)   // ... omitted
    #define IS_MOUSE_CENTER(k)  // ... omitted
    #define IS_MOUSE_SINGLE(k)  // ... omitted
    #define IS_MOUSE_DOUBLE(k)  // ... omitted
    #define IS_MOUSE_DOWN(k)    // ... omitted
    #define IS_MOUSE_UP(k)      // ... omitted
```

The `IS_MOUSE_KEY()` macro returns a nonzero value if the given key code indicates a mouse event. The `IS_TRUE_KEY()` macro returns a nonzero value if the given key code indicates a keyboard key. The `IS_EXT_ASCII_KEY()` macro returns a nonzero value if the given key code represents a character that can appear in a buffer (rather than a function key or cursor key). The `IS_WIN_KEY()` macro returns a nonzero value if the given key code indicates a window event like a menu selection, pressing a button on a dialog, or getting the focus, while `IS_WIN_PASSIVE_KEY()` returns nonzero if the given key represents an incidental event: losing or gaining focus, losing the selection, or the mouse entering or leaving Epsilon's window.

The `IS_MOUSE_LEFT()`, `IS_MOUSE_RIGHT()`, and `IS_MOUSE_CENTER()` macros return nonzero if a particular key code represents either a single or a double click of the indicated button. The `IS_MOUSE_SINGLE()` and `IS_MOUSE_DOUBLE()` macros return nonzero if the given key code represents a single-click or double-click, respectively, of any mouse button. The `IS_MOUSE_DOWN()` macro returns nonzero if the key code represents the pressing of any mouse button (either a single-click or a double-click). Finally, the `IS_MOUSE_UP()` macro tells if a particular key code represents the release of any mouse button.

```
    user short mouse_pixel_x, mouse_pixel_y;
    int y_pixels_per_char()
    int x_pixels_per_char()
    clip_mouse()  /* mouse.e subr. */
```

On most systems, Epsilon can provide the mouse position with finer resolution than simply which character it is on. The `mouse_pixel_x` and `mouse_pixel_y` variables contain the mouse position in the most accurate form Epsilon provides. Setting the pixel variables moves the mouse cursor and resets the `mouse_x` and `mouse_y` variables to match. Similarly, setting `mouse_x` or `mouse_y` resets the corresponding pixel variable.

EEL subroutines should not assume any particular scaling between the screen character coordinates provided by `mouse_x` and `mouse_y` and these "pixel" variables. The scaling varies with the screen display mode or selected font. As with the character coordinates, the upper left corner has pixel coordinate $(0, 0)$. The `y_pixels_per_char()` and `x_pixels_per_char()` primitives report the current scaling between pixels and characters. For example, `mouse_x` usually equals the quantity `mouse_pixel_x / x_pixels_per_char()`, rounded down to an integer.

The `mouse_x` variable can range from -1 to `screen_cols`, while the valid screen columns range from 0 to (`screen_cols - 1`). Epsilon uses the additional values to indicate that the user has tried to move the mouse cursor off the screen, in environments which can detect this (currently, no supported environments can). The `mouse_pixel_x` variable, on the other hand, ranges from 0 to `screen_cols * x_pixels_per_char()`. The highest and lowest values of `mouse_pixel_x` correspond to the highest and lowest values of `mouse_x`, while other values obey the relation outlined in the previous paragraph. The `mouse_y` and `mouse_pixel_y` variables work in the same way.

The `clip_mouse()` subroutine alters the `mouse_x` and `mouse_y` variables so that they refer to a valid screen column, if they currently range off the screen.

```
    user short mouse_shift;
    short shift_pressed()
```

```
#define KB_ALT_DN     0x08    // Some Alt key
#define KB_CTRL_DN    0x04    // Some Ctrl key
#define KB_LSHIFT_DN  0x02    // Left shift key
#define KB_RSHIFT_DN  0x01    // Right shift key
#define KB_SHIFT_DN   (KB_LSHIFT_DN | KB_RSHIFT_DN)
                              // Either shift key

int was_key_shifted()
```

When Epsilon dequeues a mouse event with `wait_for_key()`, it also sets the `mouse_shift` variable to indicate which shift keys were depressed at the time the mouse event was enqueued. The `shift_pressed()` primitive returns the same codes, but indicates which shift keys are depressed at the moment you call it.

The `was_key_shifted()` subroutine tells if the user held down Shift when pressing the current key. Some keys produce the same key code with or without shift.

Unlike the `shift_pressed()` primitive, which reports on the current state of the Shift key, this one works with keyboard macros by returning the state of the Shift key at the time the key was originally pressed. A subroutine must call `was_key_shifted()` at the time the macro is recorded for the Shift state to be recorded in the macro. Macros defined by a command file can use an `E-` prefix to indicate this.

```
short mouse_buttons()
int mouse_pressed()
get_movement_or_release()  /* menu.e */
```

The `mouse_buttons()` primitive returns the number of buttons on the mouse. A value of zero means that Epsilon could not find a mouse on the system.

The `mouse_pressed()` primitive returns a nonzero value if and only if some button on the mouse has gone down but has not yet gone up. The subroutine `get_movement_or_release()` uses this function. It delays until the mouse moves or all its buttons have been released.

Mouse Cursors

```
user short mouse_display;
user short mouse_auto_on;    /* default = 1 */
user short mouse_auto_off;   /* default = 1 */
```

The `mouse_display` primitive controls whether or not Epsilon displays the mouse cursor. Set it to zero to turn the mouse cursor off, and to a nonzero value to turn the mouse cursor on. Turning off the mouse cursor does not cause Epsilon to stop queuing up mouse events—to do that, use `catch_mouse`.

Epsilon automatically turns on the mouse cursor when it detects mouse motion, if the `mouse_auto_on` primitive has a nonzero value. Epsilon automatically turns off the mouse when you start to type on the keyboard, if the `mouse_auto_off` primitive has a nonzero value. Neither of these actions affect the status of queuing up mouse events. When Epsilon automatically turns on the mouse cursor, it sets `mouse_display` to 2.

```
typedef struct mouse_cursor {
        byte on_pixels[32];
        byte off_pixels[32];
        byte hot_x, hot_y;
```

```
        short stock_cursor;
} MOUSE_CURSOR;
MOUSE_CURSOR *mouse_cursor;
MOUSE_CURSOR std_pointer;
```

You can select a different mouse cursor in Epsilon for Windows by setting the `mouse_cursor` primitive. It points to a structure of type `MOUSE_CURSOR`. The `MOUSE_CURSOR` type is built into Epsilon. In the current version, only the `stock_cursor` member is used. It selects one of several standard Windows cursors, according to the following table, which lists the stock cursor codes defined in codes.h:

`CURSOR_ARROW`	Standard arrow
`CURSOR_IBEAM`	Text I-beam
`CURSOR_WAIT`	Hourglass
`CURSOR_CROSS`	Crosshair
`CURSOR_UPARROW`	Arrow pointing up
`CURSOR_SIZE`	Resize
`CURSOR_ICON`	Empty icon
`CURSOR_SIZENWSE`	Double-headed arrow pointing northwest and southeast
`CURSOR_SIZENESW`	Double-headed arrow pointing northeast and southwest
`CURSOR_SIZEWE`	Double-headed arrow pointing east and west
`CURSOR_SIZENS`	Double-headed arrow pointing north and south
`CURSOR_PAN`	Neutral cursor for wheeled mouse panning
`CURSOR_PAN_UP`	Wheeled mouse cursor when panning up
`CURSOR_PAN_DOWN`	Wheeled mouse cursor when panning down

The `std_pointer` primitive variable contains Epsilon's standard left-pointing arrow cursor. Use the syntax `mouse_cursor = &some_cursor;` to set the cursor to a different `MOUSE_CURSOR` variable. In Epsilon for Unix under X, the panning cursors above are available, but not the others.

Mouse Subroutines

```
window int (*mouse_handler)();
allow_mouse_switching(int nwin)     // mouse.e subr.
buffer char mouse_dbl_selects;
char run_by_mouse;
char show_mouse_choices;
```

The mouse.e and menu.e files define the commands and functions normally bound to the mouse buttons. The functions that handle button clicks examine the window-specific function pointer `mouse_handler` so that you can easily provide special functions for clicks in a particular window. By default, the variable contains 0 in each window, so that Epsilon does no special processing. Set the variable to point to a function, and Epsilon will call it whenever the user pushes a mouse button and the mouse cursor is over the indicated window. The function receives one parameter, the window handle of the specified window. It can return nonzero to prevent the normal functioning of the button, or zero to let the function proceed.

The `allow_mouse_switching()` subroutine is a `mouse_handler` function. Normally, when a pop-up window is on the screen, Epsilon doesn't let the user simply switch to another window. Depending on the context, Epsilon either removes the pop-up window and then switches to the new window, or signals an error

and remains in the pop-up window. If you set the `mouse_handler` variable in a particular window to the `allow_mouse_switching()` subroutine, Epsilon will permit switching to that window if the user clicks in it, without deleting any pop-up window.

The buffer-specific `mouse_dbl_selects` variable controls what double-clicking with a mouse button does. By default the variable is zero, and double-clicking selects words. If the variable is nonzero, Epsilon instead runs the command bound to the ⟨Newline⟩ key.

The `run_by_mouse` variable is normally zero. Epsilon sets it to one while it runs a command that was selected via a pull-down menu or using the tool bar. Commands can use this variable to behave differently in this case. For example, the subroutine that provides completion automatically produces a list of choices to choose from, when run via the mouse. It does this if the `MUST_MATCH` flag (see page 315) indicates that the user must always pick one of the choices (instead of typing in a different selection), or if the `show-mouse-choices` variable is nonzero.

The Scroll Bar

```
user window int display_scroll_bar;
int scroll_bar_line()
```

The built-in variable `display_scroll_bar` controls whether or not the current window's right border contains a scroll bar. Set it to zero to turn off the scroll bar, or to any positive number to display the bar. If a window has no right border, or has room for fewer than two lines of text, Epsilon won't display a scroll bar. Although the EEL functions that come with Epsilon don't support clicking on a scroll bar on the left border of a window, Epsilon will display one if `display_scroll_bar` is negative. Any positive value produces the usual right-border scroll bar. (This variable, and the following primitive, have no effect in Epsilon for Windows, which handles scrolling internally.)

The `scroll_bar_line()` primitive returns the position of the scroll box diamond on the scroll bar. A value of one indicates the line just below the arrow at the top of the scroll bar. Epsilon always positions this arrow adjacent to the first line of text in the window, so a return value of *n* indicates the scroll box lies adjacent to text line *n* in the window (numbered from zero).

```
scroll_by_wheel(int clicks, int per_click)
```

When you use a wheeled mouse like the Microsoft IntelliMouse, Epsilon calls the `scroll_by_wheel()` subroutine whenever you roll its wheel. (See the next section for information on what happens when you click the wheel, not roll it.) Epsilon provides the number of clicks of the wheel since the last time this function was called (which may be positive or negative) and the control panel setting that indicates the number of lines Epsilon should scroll on each click.

After calling this subroutine, Epsilon can then optionally generate a `WIN_WHEEL_KEY` key event. See page 312.

Mouse Panning

```
int mouse_panning;
int mouse_panning_rate(int percent, int slow, int fast)
```

The `mouse_panning` variable and the `mouse_panning_rate()` primitive work together to support panning and auto-scroll with the Microsoft IntelliMouse (or any other three button mouse). The EEL

subroutine that receives clicks of the third mouse button sets `mouse_panning` nonzero to tell Epsilon to begin panning and record the initial position of the mouse.

Then the subroutine can regularly call `mouse_panning_rate()` to determine how quickly, and in what direction, to scroll. The parameter `percent` specifies the percentage of the screen the mouse has to travel to reach maximum speed (usually 40%). The parameter `slow` specifies the minimum speed in milliseconds per screen line (usually 2000 ms/line). The parameter `fast` specifies the maximum speed in milliseconds per screen line (usually 1 ms/line).

The `mouse_panning_rate()` primitive uses these figures, plus the current position of the mouse, to return the scroll rate in milliseconds per screen line. It returns a positive number if Epsilon should scroll down, a negative number to scroll up, or zero if Epsilon should not scroll.

See the previous section for information on what happens when you roll the wheel on a wheeled mouse instead of clicking it.

8.6.3   Window Events

When an EEL function calls `getkey()` to retrieve the next key, it sometimes receives a key code that doesn't correspond to any actual key, but represents some other kind of input event. Mouse keys (see page 307) are one example of this. This section describes the other key codes Epsilon uses for input events. These keys only occur in the Windows version.

The `WIN_MENU_SELECT` key indicates that the user selected an item from a menu or the tool bar. Epsilon sets the variable `menu_command` to the name of the selected command whenever it returns this key.

The `WIN_DRAG_DROP` key indicates that the user has just dropped a file on one of Epsilon's windows, or that Epsilon has received a DDE message from another program. See the description of the `drag_drop_result()` primitive on page 274.

The `WIN_EXIT` key indicates that the user has tried to close Epsilon, by clicking on the close box, for example.

The `WIN_HELP_REQUEST` key indicates that the user has just pushed a button in Epsilon's help file to set a particular variable or run a command. Epsilon fills the `menu_command` variable with the message from the help system.

The `GETFOCUS` and `LOSEFOCUS` keys indicate that a particular screen has gained or lost the focus. These set `mouse_screen` just like mouse keys. (See page 307.)

The `WIN_RESIZE` key indicates that Epsilon has resized a screen. Sometimes Epsilon will resize the screen without returning this key.

The `WIN_VERT_SCROLL` key indicates that Epsilon has scrolled a window. Epsilon doesn't normally return keys for these events. Instead, Epsilon calls the EEL subroutine `scrollbar_handler()` from within the `wait_for_key()` function, passing it information on which scroll bar was clicked, which part of the scroll bar was selected, and so forth.

Epsilon only recognizes user attempts to scroll by clicking on the scroll bar, or to resize the window, when it waits for a key in a recursive edit level. When an EEL command requests a key, Epsilon normally ignores attempts to scroll, and postpones acting on resize attempts.

An EEL command can set the `permit_window_keys` variable to allow these things to happen immediately, and possibly redraw the screen. Bits in the variable control these activities: set the `PERMIT_SCROLL_KEY` bit to permit immediate scrolling, and set `PERMIT_RESIZE_KEY` to permit resizing. Setting `PERMIT_SCROLL_KEY` also makes Epsilon return the `WIN_VERT_SCROLL` key shortly after scrolling. Setting the `PERMIT_WHEEL_KEY` bit tells Epsilon to generate a `WIN_WHEEL_KEY` key event after scrolling due to a wheel roll on a Microsoft IntelliMouse.

The WIN_BUTTON key indicates that the user has clicked on a button in a dialog box, or selected the button via the keyboard. By default, Epsilon translates many buttons to standard keys like Ctrl-M. An EEL program can set the variable return_raw_buttons to disable this translation and instead receive WIN_BUTTON keys for each button pressed. For other buttons, and for check boxes and certain other dialog events, Epsilon always enqueues a WIN_BUTTON key. For each of these input events, it sets the key_is_button variable to a distinct value.

8.6.4 Completion

There are several EEL subroutines defined in complete.e that get a line of input from the user, allowing normal editing. Most of them offer some sort of completion as well. They also provide a command history.

Each function takes two or three arguments. The first argument is an array of characters in which to store the result. The second argument is a prompt string to print in the echo area. The third argument, if there is one, is the default string. Depending on the setting of the insert-default-response variable, Epsilon may insert this string after the prompt, highlighted, or it may be available by pressing Ctrl-R or Ctrl-S.

Some functions will substitute the default string if you press ⟨Enter⟩ without typing any response. These functions display the default to you inside square brackets [ ] (whenever they don't actually pre-type the default after the prompt). The prompt that you must provide to these functions shouldn't include the square brackets, or the colon and space that typically ends an Epsilon prompt. The function will add these on before it displays the prompt. If there should be no default, use the empty string "".

```
get_file(char *res, char *pr, char *def)
get_file_dir(char *res, char *pr)
```

The get_file() and get_file_dir() subroutines provide file name completion. When the get_file() subroutine constructs its prompt, it begins with the prompt string pr, then appends a colon ':' and a space. (If insert-default-response is zero, it also includes the default value in the prompt, inside square brackets.) If the user presses ⟨Enter⟩ without typing any response, get_file() copies the default def to the response string res.

The get_file_dir() subroutine provides the directory part of the current file name, inserted as part of a default response or available via Ctrl-S or Ctrl-R (see the description of the prompt-with-buffer-directory variable), but it doesn't display that as part of the prompt. It uses the prompt pr as is. It doesn't substitute any default if the user enters no file name. Both get_file() and get_file_dir() call absolute() on the name of the file before returning (see page 262).

```
get_buf(char *res, char *pr, char *def)
```

The get_buf() subroutine completes on the name of a buffer. To construct its prompt, the subroutine begins with the prompt string pr, then adds the default def inside square brackets [ ], and then appends a colon ':' and a space.

```
get_any(char *res, char *pr, char *def)
get_cmd(char *res, char *pr, char *def)
get_macname(char *res, char *pr, char *def)
get_func(char *res, char *pr, char *def)
get_var(char *res, char *pr, char *def, int flags)
```

Epsilon locates commands, subroutines, and variables by looking them up in its *name table*. See page 291 for details. The subroutines that complete on commands, variables and so forth all look in the same

table, but restrict their attention to particular types of name table entries. For example, the `get_macname()` subroutine ignores all name table entries except those for keyboard macros. In the following table, * indicates that the subroutine allows entries of that type.

	Command	Subr.	Kbd. Macro	Key Table	Variable
`get_any()`	*	*	*	*	*
`get_cmd()`	*		*		
`get_func()`	*	*			
`get_macname()`			*		
`get_var()`					*

These subroutines all substitute the default string if you just press ⟨Enter⟩ without entering anything. They also display the default inside square brackets [ ] after the prompt you provide (if `insert-default-response` is zero), and then append a colon ':' and a space.

The `get_var()` subroutine takes an additional, fourth parameter. It contains a set of flags to pass to the `comp_read()` subroutine, as listed below.

```
int get_command_index(char *pr)
```

The `get_command_index()` subroutine defined in control.e calls the `get_cmd()` subroutine to ask the user for the name of a command. It then checks to see if the command exists, and reports an error if it doesn't. (When checking, it allows subroutines and macros as well as actual commands.) If the function name checks out, `get_command_index()` returns its name table index.

```
int get_key_response(char *valid, int def)
```

The `get_key_response()` subroutine waits for the user to type a valid key in response to a prompt. The parameter `valid` lists the acceptable characters, such as `"YN"` for a yes/no question. (But see the `ask_yn()` subroutine, more suitable for yes/no questions.) The `def` parameter, if greater than zero, indicates which key should be the default if the user presses ⟨Enter⟩. The subroutine returns the selected key.

Completion Internals

```
    /* bits for finder func */
#define STARTMATCH      1
#define LISTMATCH       2
#define EXACTONLY       4
#define FM_NO_DIRS      (0x10)
#define FM_ONLY_DIRS    (0x20)
char *b_match(char *partial, int flags)
              /* sample finder */

comp_read(char *response, char *prmpt,
          char *(*finder)(), int flags, char *def)

    /* bits for comp_read() */
#define CAUTIOUS        (0x100)
#define COMP_FOLD       (0x200)
#define MUST_MATCH      (0x400)
```

```
#define NONE_OK         (0x800)
#define POP_UP_PROMPT   (0x1000)
#define COMP_FILE       (0x2000 | CAUTIOUS)
#define PASSWORD_PROMPT (0x4000)
#define SPACE_VALID     (0x8000)


prompt_comp_read(char *response, char *prmpt,
                 char *(*finder)(), int flags,
                 char *def)
zeroed char completion_column_marker;
```

It's easy to add new subroutines that can complete on other things. First, you must write a "finder" function that returns each of the possible matches, one at a time, for something the user has typed. For example, the `get_buf()` subroutine uses the finder function `b_match()`.

A finder function takes a parameter `partial` which contains what the user's typed so far, and a set of `flags`. If the `STARTMATCH` flag is on, the function must return the first match of `partial`. If `STARTMATCH` is off, it should return the next match. The function should return 0 when there are no more matches. The `LISTMATCH` flag is on when Epsilon is preparing a list of choices because the user has pressed '?'. This is so that a finder function can format the results differently in that case. If the `EXACTONLY` flag is on, the finder function should return only exact matches for `partial`. If the finder function is matching file names, you may also provide the `FM_NO_DIRS` flag, to exclude directory names, or `FM_ONLY_DIRS` to retrieve only directory names.

Next, write a subroutine like the various `get_` routines described above, all of which are defined in complete.e. It should take a prompt string, possibly a default string, and a character pointer in which to put the user's response. It passes these to the `comp_read()` subroutine, along with the name of your finder function (as a function pointer).

The `comp_read()` subroutine also takes a `flags` parameter. If the `CAUTIOUS` flag is zero, `comp_read()` assumes that all matches for a certain string will begin with that string, and that if there is only one match for a certain string, adding characters to that string won't generate any more matches. These assumptions are true for most things Epsilon completes on, but they're not true for files. (For example, if the only match for x is xyz, but xyz is a directory with many files, the second assumption would be false. The first assumption is false when Epsilon completes on wildcard patterns like `*.c`, since none of the matches will start with the `*` character.) If you provide the `CAUTIOUS` flag when you call `comp_read()`, Epsilon doesn't make those assumptions, and completion is somewhat slower.

Actually, when completing on files, provide the `COMP_FILE` macro instead of just `CAUTIOUS`; this includes `CAUTIOUS` but also makes Epsilon use some special rules necessary for completing on file names.

If you provide the `COMP_FOLD` flag to `comp_read()`, it will do case-folding when comparing possible completions.

The `MUST_MATCH` flag tells `comp_read()` that if the user types a response that the finder function doesn't recognize, it's probably a mistake. The `comp_read()` subroutine will then offer a list of possible responses, even though the user may not have pressed a key that ordinarily triggers completion. The `comp_read()` subroutine might still return with an unrecognized response, though. This flag is simply advice to `comp_read()`. The `NONE_OK` flag is used only with `MUST_MATCH`. It tells `comp_read()` that an empty response (just typing ⟨Enter⟩) is ok.

Under Epsilon for Windows, the `POP_UP_PROMPT` flag tells `comp_read()` to immediately pop up a one-line dialog box when prompting. Right now, this flag may only be used when no completion is involved, and `comp_read()` is simply prompting for a line of text.

The `PASSWORD_PROMPT` flag tells `comp_read()` to display each character of the response as a `*` character. When the Internet functions prompt for a password they use this flag.

The SPACE_VALID flag tells comp_read() that a ⟨Space⟩ character is valid in the response. Since ⟨Space⟩ is also a completion character, comp_read() tries to guess whether to add a ⟨Space⟩ or complete, by examining possible matches.

A finder function receives any of the above flags that were passed to comp_read(), so it can alter its behavior if it wants.

The comp_read() subroutine uses the prompt you supply as-is. Usually, the prompt should end with a colon and a space, like "Find file: ". By contrast, the prompt_comp_read() subroutine adds to the supplied prompt by showing the default value inside square brackets, when insert-default-response is zero. The prompt string you supply to it should not end with a colon and space, since Epsilon will add these. If you provide a prompt such as "Buffer name" and a default value of "main", Epsilon will display Buffer name [main]: . If the default value you provide is empty or too long, Epsilon will instead display Buffer name: , omitting the default. Whether or not Epsilon displays the default, if the user doesn't enter any text at the prompt the prompt_comp_read() subroutine substitutes the default value by copying def to response.

Sometimes it's convenient if a finder function returns matches with additional data after them which shouldn't be included in the returned response. If the variable completion_column_marker is nonzero and a response from the finder function contains it, Epsilon strips that character and any following text from the response before returning it.

```
char *(*list_finder)();
list_matches(char *s, char *(*finder)(), int flags, int mbuf)
int *(*completion_lister)();
char resize_menu_list;
```

The comp_read() subroutine looks at several variables whenever it needs to display a list of possible completions (such as when the user types '?'). You can change the way Epsilon displays the list by setting these variables. Typically, you would use the save_var statement to temporarily set one of these while your completion routine runs.

By default, Epsilon calls the list_matches() subroutine to prepare its buffer of possible matches. The function takes the string to complete on, the finder function to use, flags as described above, and a buffer number. It calls the finder function repeatedly (passing it the LISTMATCH flag as well as any others passed to list_matches()) and puts the resulting matches into the indicated buffer, after sorting the matches. If the completion_lister function pointer is non-null, Epsilon calls that function instead of list_matches(), passing it the same parameters. If, for example, you have to sort the matches in a special order, you can set this variable.

If you simply want a different list of matches when Epsilon lists them, as opposed to when Epsilon completes on them, you can set the list_finder function pointer to point to a different finder function. The list_matches() subroutine always uses this variable if non-null, instead of the finder function it receives as a parameter.

An EEL completion function can temporarily set the resize_menu_list variable nonzero to indicate that if the user tries to list possible completion choices, the window displaying the choices should be widened if necessary to fit the widest choice. This variable has no effect on Epsilon windows within GUI dialogs.

```
int complete(char *response, char *(*finder)(), int flags)
```

To actually do completion, comp_read() calls the complete() subroutine. It takes a finder function pointer, flags like CAUTIOUS and COMP_FOLD described above, and a string to complete on. It tries to extend the string with additional characters from the matches, modifying it in place.

The `complete()` subroutine generally returns the number of possible matches for the string. However, it may be able to determine that no more completion is possible before reaching the last match. For example, if the subroutine tries to complete on the file name "foo", and encounters files named "foobar", "foobaz", "foo3", "foo4" and so forth, it can determine on the third file that no completion is possible. In this case, it returns 3, even though there may be additional matches. It can only "give up early" in this way when it has encountered two or more matches. So when the subroutine returns a value of two or greater, there may be additional matches not included in its count.

```
build_prompt(char *full, char *pr, char *def, int omit, int rel)
```

The `build_prompt()` subroutine helps construct the text of a prompt. It copies the prompt *pr* to *full*, appending the default value *def* to it (inside brackets).

If the combination would be too wide for the screen, the subroutine abbreviates the default value. If even an abbreviated value would be too wide, or if *omit* is nonzero, it omits the default from the prompt entirely. If *rel* is nonzero, it assumes *def* is an absolute pathname, and uses its relative form.

```
find_buffer_prefix(int buf, char *prefix)
```

The `find_buffer_prefix()` subroutine looks through all the lines in the buffer `buf` to see if they all start with the same string of characters. It puts any such common prefix shared by all the lines in `prefix`. For instance, if the buffer contains three lines "waters", "watering" and "waterfall", it would put the string "water" in `dest`.

```
char *general_matcher(char *s, int flags)
```

Epsilon providers a general-purpose finder function called `general_matcher()`. An EEL function can perform completion on some arbitrary list of words by putting the list of words in a buffer named `_MATCH_BUF` (a macro defined in eel.h) and then providing `general_matcher()` as a finder function to a subroutine like `comp_read()`. Call `comp_read()` with the `COMP_FOLD` flag if you want `general_matcher()` to ignore case when comparing.

```
char _doing_input;
keytable comp_tab, menu_tab, view_tab;
```

An EEL function can tell if Epsilon is currently prompting for a line of input (or performing certain related tasks) by examining the `_doing_input` variable. It's zero normally. While Epsilon is inside the `search_read()` subroutine or related ones, getting a search string from the user, it's set to `DI_SEARCH`. While Epsilon is prompting for a line of some other type of input, it's set to `DI_LINEINPUT`.

While the `view_buf()` subroutine (or a related one) is displaying a pop-up window, `_doing_input` is set to either `DI_VIEW` or `DI_VIEWLAST`, according to whether the `last` parameter passed to `view_buf()` was zero or not.

When Epsilon prompts for input, it uses certain specialized key tables in the buffer that accepts the input. It uses the `comp_tab` key table for the one-line buffer where the user types a response, in all the subroutines that accept a line of input (except in `search_read()` and related subroutines that prompt for search strings).

When Epsilon displays a list of possible matches, or previous responses, or similar things, while getting a line of input, it uses the `menu_tab` key table in the buffer displaying the list.

Finally, subroutines like `view_buf()` normally use the `view_tab` key table to show a buffer in a pop-up window.

Listing Commands, Buffers, or Files

```
int name_match(char *prefix, int start)
```

Several primitives help to perform completion. The `name_match()` primitive takes a command prefix such as "nex" and a number. It finds the next command or other name table entry that begins with the supplied prefix, returning its name table index. If its numeric argument is nonzero, it starts at the beginning of the name table. Otherwise it continues from the name table index returned on the previous call. It returns zero when there are no more matching names. When comparing names, case doesn't count and '-' is the same as '_'.

```
char *buf_match(char *pattern, int flags)
char *do_file_match(char *pattern, int flags)
#define STARTMATCH      1
#define EXACTONLY       4
#define FM_NO_DIRS     (0x10)
#define FM_ONLY_DIRS   (0x20)
#define FM_FOLD        (0x200)
char *file_match(char *pattern, int flags)
```

The `buf_match()` and `file_match()` primitives are similar to `name_match()`. Instead of returning a command index, they return the actual matching buffer or file names, respectively, and return a null pointer when there are no more matches.

The `buf_match()` primitive returns one of a series of buffer names that match a pattern. The pattern is of the sort that `fpatmatch()` accepts: * matches any number of characters, ? matches a single character, and [a-z] represents a character class. The `STARTMATCH` flag tells it to examine the pattern and return the first match; omitting the flag makes it return the next match of the current pattern. The `EXACTONLY` flag tells it to return only exact matches of the pattern; otherwise it returns buffer names that start with a match of the pattern (as if it ended in *). The `FM_FOLD` flag tells it to ignore case when comparing buffer names against the pattern; by default buffer names are case-sensitive.

The `file_match()` primitive returns one of a series of file names that match a pattern. You can use this primitive to expand file name patterns such as a*.c. See page 108 for details on Epsilon's syntax for file patterns. The `STARTMATCH` flag tells it to examine the pattern and return the first match; omitting the flag makes it return the next match of the current pattern. The `EXACTONLY` flag tells it to return only exact matches of the pattern; otherwise it returns file names that start with a match of the pattern. Use the `FM_NO_DIRS` flags if you want to skip over directories when looking for files that match, or `FM_ONLY_DIRS` to retrieve only directory names.

Instead of directly calling the `file_match()` primitive, you should call the subroutine `do_file_match()`. It takes the same arguments as `file_match()` and returns the same value. In fact, by default it simply calls `file_match()`. But a user extension can replace the subroutine to provide Epsilon with new rules for file matching.

```
short abort_file_matching = 0;
#define ABORT_IGNORE  0   /* ignore abort key & continue */
#define ABORT_JUMP    -1  /* jump via check_abort() */
#define ABORT_ERROR   -2  /* return ABORT_ERROR as error code */
```

By default, the `file_match()` and `do_dired()` primitives ignore the abort key. (See page 261 for information on `do_dired()`.) To permit aborting a long file match, set the primitive variable

`abort_file_matching` using `save_var` to tell Epsilon what to do when the user presses the abort key. If you set `abort_file_matching` to `ABORT_ERROR` and the user presses the abort key, this function will return a failure code and set `errno` to `EREADABORT`. Set the variable to `ABORT_JUMP` if you want Epsilon to abort your function by calling the `check_abort()` primitive. (See page 284.) By default, the variable is zero, and Epsilon ignores the abort key until the primitive finishes.

8.6.5 Other Input Functions

```
get_strdef(char *res, char *pr, char *def)
get_strnone(char *res, char *pr, char *def)
get_string(char *res, char *pr)
get_str_auto_def(char *res, char *pr)
get_strpopup(char *res, char *title,
             char *def, char *help)
```

The subroutines `get_string()`, `get_strdef()`, and the rest each get a string from the user, and perform no completion. They each display the prompt, and accept a line of input with editing.

The `get_strdef()` routine additionally displays the default string (indicated by `def`) and allows the user to select the default by typing just the ⟨Enter⟩ key. The user can also pull in the default with Ctrl-S, and then edit the string if desired. While the other two functions use their prompt arguments as-is, `get_strdef()` constructs the actual prompt by adding a colon and space. If `insert-default-response` is zero, they also include the default value in the prompt, inside square brackets.

The `get_strnone()` subroutine works like `get_strdef()`, except that the default string is not displayed in the prompt (even when `insert-default-response` is zero), and Epsilon won't replace an empty response with the default string. Use this instead of `get_strdef()` if an empty response is valid.

The `get_str_auto_def()` subroutine is like `get_strdef()`, except it automatically provides the last response to the current prompt as a default.

The `get_strpopup()` subroutine is a variation of `get_strnone()` that is only available under Epsilon for Windows. It displays a simple dialog. The parameter `title` provides the dialog's title, and `def` provides the initial contents of the response area, which is returned in `res`. If the user presses the Help button, Epsilon will look up help for the specified command or variable name or other topic name in its help file.

```
int get_number(char *pr)
int numtoi(char *str)
int strtoi(char *str, int base)
char got_bad_number;
```

The `get_number()` subroutine is handy when a command needs a number. It prompts for the number using `get_string()`, but uses the prefix argument instead if one is provided. It returns the number obtained, and also takes care of resetting `iter` if necessary. It also understands numbers such as `0x10` in EEL's hexadecimal (base 16) format, binary and octal numbers, and character codes like `'a'`.

The `get_number()` subroutine uses the `numtoi()` subroutine to convert from the typed string to a number. The `numtoi()` subroutine skips over any spaces at the beginning of its string parameter, determines the base (by seeing if the string starts with "0x" or similar), and then calls `strtoi()` to perform the actual conversion. The subroutine `strtoi()` takes a string and a base, and returns the value of the string assuming it is a number in that base. It handles bases from 2 to 16, and negative numbers too. It stops when it finds a character that is not a legal digit in the requested base. Both `numtoi()` and `strtoi()` are defined in basic.e.

The subroutines `get_number()`, `numtoi()`, and `strtoi()` set the variable `got_bad_number` to a nonzero value if the string they receive doesn't indicate a valid number. They return the value zero in this case. If the string does represent a number, they set `got_bad_number` to zero.

```
int get_choice(int list, char *resp, char *title,
               char *msg, char *b1, char *b2,
               char *b3)
int select_menu_item(int resbuf, int menuwin,
                     int owin, int dir)
```

The `get_choice()` subroutine provides a way to ask the user to select one of a list of choices. The choices must appear in the buffer `list`, one to a line. The subroutine displays a pop-up window with the indicated title and shows the specified message.

Epsilon for Windows instead displays a dialog with the indicated title, and doesn't use the message. It uses the specified button labels (see the description of the `button_dialog()` primitive on page 322 for details). The `get_choice()` subroutine puts the user's choice in `resp` and returns 1. If the user cancels, the subroutine returns 0.

If `resp` is initially nonempty, `get_choice()` will position point on the first line starting with that text. If `resp` is initially `""`, the subroutine won't change point in `list`.

The `get_choice()` subroutine uses the `select_menu_item()` subroutine to handle user interaction. It takes the window handle `menuwin` of a window containing a list of choices and returns when the user has selected one. The parameter `owin` should be the handle of the window that was current before displaying `menuwin`. If `resbuf` is nonzero, Epsilon will copy the selected line to the specified buffer.

The parameter `dir` tells Epsilon how to behave when the user presses self-inserting keys like 'a'. If `dir` is zero, the subroutine interprets N and P to move forward and back, and Q to quit. Other normal keys are ignored. If `dir` is 1 or −1, and `search-in-menu` is nonzero, normal keys are added to the result, and Epsilon searches for the first (if 1) or last (if −1) item that matches.

8.6.6   Dialogs

Standard Dialogs

```
short common_file_dlg(char *fname, char *title,
                      int *flags, int save,
                      ?char *filt_str, ?char *cust_filter,
                      ?int *filt_index)
short use_common_file_dlg(char *fname, char *title,
                          int *flags, int save)
int use_common_file_dialog()
```

Under Windows, the `common_file_dlg()` primitive displays the Common Open/Save File Dialog. The `fname` parameter should be initialized to the desired default file name; on return it will hold the file name the user selected. The `title` parameter specifies the title of the dialog window. Epsilon passes the `flags` parameter to Windows; definitions for useful flag values appear in codes.h. Windows modifies some of the flags before it returns from the dialog. If the parameter `save` is nonzero, Epsilon displays the Save dialog, if zero it uses the Open dialog. This primitive uses the `common-open-curdir` variable to hold the directory that this dialog should display.

The filter parameters let you specify the file types the user can select; these are all passed directly to Windows. Epsilon normally invokes `common_file_dlg()` through the `use_common_file_dlg()`

subroutine, which uses the filter definitions from the file filter.txt to construct the `filt_str` parameter. You can edit that file to add new filters.

The parameter `filt_str` has the following format. It consists of pairs of null-terminated strings. The first string says what to display in the dialog, while the second is a Windows-style list of file patterns, separated by semicolons. For example, the first string might be `"Fortran files"` and the second string might be `"*.for;*.f77"`. A final null character follows the last pair.

The `use_common_file_dialog()` subroutine examines the `want-common-file-dialog` variable and other settings and tells whether a command should use the common file dialog in place of Epsilon's traditional file dialog.

```
find_dialog(int show)
find_dialog_say(char *text)
```

Under Windows, the `find_dialog()` primitive displays a find/replace dialog, when its parameter `show` is nonzero. When its parameter show is zero, it hides the dialog. While a find/replace dialog is on the screen, the `getkey()` function returns certain predefined keys to indicate dialog events such as clicking a button or modifying the search string. The `_find()` subroutine defined in search.e interprets these key codes to control the dialog. The global variable `find_data` lets that subroutine control the contents of the dialog.

When a find/replace dialog is on the screen, an EEL program can display an error message in it using the `find_dialog_say()` primitive. This also adds an alert symbol to the dialog. To clear the message and remove the alert symbol, pass a parameter of `""`.

```
short window_lines_visible(int w)
```

The `window_lines_visible()` primitive returns the number of lines of a given window that are visible above a find/replace dialog. If the given window contains twelve lines, but a find/replace dialog covers the bottom three, this function would return nine. If Epsilon isn't displaying a find/replace dialog, the function returns the number of lines in the given window.

```
int comm_dlg_color(int oldcolor, char *title)
```

In Epsilon for Windows, the `comm_dlg_color()` primitive lets the user select a color using the Windows common color dialog. The `oldcolor` parameter specifies the default color, and `title` specifies the dialog's title. The primitive returns the selected color, or −1 if the user canceled.

```
about_box()
```

The `about_box()` primitive displays Epsilon's "About" box under Windows. In other versions of Epsilon, it inserts similar information into the current buffer. The about-epsilon command uses this primitive.

Button Dialogs

```
short button_dialog(char *title, char *question,
                    char *yes, char *no, char *cancel,
                    int def_button)
```

The `button_dialog()` primitive displays a dialog having one to three buttons. By convention, these buttons have meanings of "Yes", "No", and "Cancel", but the labels may have any text. Set the `cancel` parameter to `""` to use a dialog with two buttons. Set both `cancel` and `no` to `""` if you want a dialog with one button. Put & before a character in a button label to make it an access key; it will be underlined, and pressing the key will act like clicking that button. Use && for a literal & character. The parameter `title` specifies the title of the dialog. The parameter `question` holds the text to display in the dialog next to the buttons.

Set `def_button` to 1, 2, or 3 to make the default button be the first, second or third. Any other value for `def_button` is the same as 1. Canceling or closing the dialog is equivalent to pressing the last defined button.

The primitive returns 1, 2, or 3 to indicate which button was pressed. This primitive only works in the Windows version of Epsilon; read on for similar functions that work everywhere.

```
int ask_yn(char *title, char *question, char *yes_button,
           char *no_button, int def_button)
ask_3way(char *title, char *question, char *prompt,
         char *a1, char *a2, char *a3, int def)
```

The `ask_yn()` subroutine defined in basic.e asks a Yes/No question. Under Windows, it uses a dialog. The parameters specify the title of the dialog, the text of the question displayed in it, and the text on its two buttons (typically "Yes" and "No", but sometimes "Save" and "Cancel" or the like). Put & before a character in a button label to make it an access key; it will be underlined, and pressing the key will act like clicking that button. Use && for a literal & character.

Set `def_button` to 0 for no default, 1 to make the first choice "Yes" the default, or 2 to make the second choice "No" the default. (Under non-Windows versions, no default means that just hitting ⟨Enter⟩ won't return from this function; you must choose an option. Under Windows, no default is the same as a default of "Yes".) The function returns 1 if the user selected the first option "Yes" or 0 if the user selected the section option "No". Non-Windows versions of Epsilon only use the `question` and `def_button` parameters. They modifies the prompt to indicate the default, if any.

The `ask_3way()` subroutine is similar, but asks a question with three possible responses. Epsilon always displays the prompt; GUI versions also display a dialog with the specified title and question. The first letter of each answer is its shortcut key; put an ampersand "&" before another letter in an answer to use that as the shortcut key instead. The value `def` may be 1, 2, or 3 to make that answer the default, or 0 for none. The subroutine returns 1, 2, 3 as its response, or 0 if the user aborted.

Windowed Dialogs

```
display_dialog_box(char *dialogname, char *title,
                   int win1, int win2, int win3,
                   char *button1, char *button2, char *button3,
                   char *button4, ?char *tag)
```

The `display_dialog_box()` primitive creates a new dialog box in Epsilon for Windows containing one or more Epsilon windows. The `dialogname` must correspond to one of the dialogs in this list:

Dialog name	Windows	Dialog name	Windows
AskExitBox	2	GeneralBox	1
AskSaveBox	2	HelpSetup1	1
CaptionBox	2	OneLineBox	1
EditVarBox	2	PromptBox	2
FileDateBox	1	SetColorBox	3
FileDateBox2	1	UsageBox	1

Each dialog requires one to three handles to pop-up windows, created with `add_popup()` in the usual way. The primitive moves these windows to the new dialog box. If you use a dialog which requires only one or two window handles, provide zero for the remaining handles. The windows will be resized to fit the dialog, and each will be assigned a unique "screen handle". Mouse clicks in that window will set the `mouse_screen` variable to the matching screen handle. You can use the `window_to_screen()` primitive to determine the screen number assigned to each window.

The parameters `button1`, `button2`, `button3`, and `button4` specify the text for the buttons. If you want fewer buttons, provide the value `""` for any button except button 1 and it will not appear. The specified `title` appears at the top of the dialog box.

When you click on a button in a dialog, Epsilon normally returns a particular fixed keystroke: either Ctrl-M, or the abort key specified by the `abort_key` variable, or the help key specified by the `HELPKEY` macro, for the first, second, and third buttons respectively. These correspond to typical button labels of "OK", "Cancel", and "Help", so that most EEL programs don't need to do anything special to receive input from buttons. If an EEL program needs to know whether a keypress came from an actual key, or a button, it can examine the value of the `key_is_button` variable. This variable is zero whenever the last key returned was an actual key, and nonzero when it was really a button. In the latter case, its value is `1` if the leftmost button was pressed, `2` if the next button was pressed, and so forth.

Sometimes an EEL program puts different labels on the buttons. It can be more convenient in this case to retrieve a button press as a distinct key. Set the `return_raw_buttons` variable to a nonzero value to retrieve all button presses as the key code `WIN_BUTTON`. The `key_is_button` variable will still be set as described above, so you can distinguish one button from another by examining its value.

Epsilon for Windows remembers and restores the sizes of most of these dialogs. Some dialogs may be used in multiple contexts. To have Epsilon remember a different size for each context, pass a unique `tag` parameter. If the optional `tag` is missing or `NULL`, Epsilon uses a default context for remembering the dialog's size.

```
one_window_to_dialog(char *title, int win1,
               char *button1, char *button2, char *button3)
prompt_box(char *title, int win1, int win2, char *tag)
two_scroll_box(char *title, int win1, int win2,
               char *button1, char *button2, char *button3)
void (*use_alternate_dialog)(int win1, int win2, int win3);
char *use_alternate_dialog_name;
```

The subroutines `one_window_to_dialog()`, `prompt_box()`, and `two_scroll_box()` each call `display_dialog_box()` with some of its parameters filled in for you. They display certain common kinds of dialogs. Call `one_window_to_dialog()` to display a dialog with a single text window and one to three buttons. To see an example, define a bookmark with Alt-/ and then type Alt-X list-bookmarks. Call `prompt_box()` to display a dialog with a one-line window, and below it a list-box style window. To see an example, type Ctrl-X Ctrl-F and then '?'. Call `two_scroll_box()` to display a dialog box with two multi-line windows.

These subroutines all call a subroutine named `do_display_dialog_box()`, which takes the same parameters as `display_dialog_box()`, but can be told to use an alternative function to display the dialog, not `display_dialog_box()`, by temporarily setting the function pointer `use_alternate_dialog` to a suitable function. Or they can be told to use a different dialog name by temporarily setting the `use_alternate_dialog_name` variable to its name.

```
next_dialog_item()
prev_dialog_item()
```

Within an Epsilon window that's part of a dialog box, the `next_dialog_item()` and `prev_dialog_item()` primitives move the focus to a different window or button within the dialog box. Epsilon normally binds ⟨Tab⟩ and Shift-⟨Tab⟩ to commands that use these primitives.

```
int dialog_checkboxes;
```

Some dialogs include check boxes. The `dialog_checkboxes` variable controls which check boxes are checked. Each check box corresponds to a bit in this variable. When EEL code sets or clears a bit, it's reflected in the state of that check box; similarly, when the user clicks the check box, its corresponding bit in this variable changes to match. This also generates a `WIN_BUTTON` event; see page 313.

```
set_window_caption(int win, char *title)
show_window_caption()
```

The `set_window_caption()` primitive sets the text in the title bar of the dialog box containing the window `win`. If the specified window is on Epsilon's main screen, it sets the main window title displayed above the menu bar. The `show_window_caption()` subroutine calls this to include the current file name in the caption of Epsilon's main window.

8.6.7   The Main Loop

While Epsilon runs, it repeatedly gets keys, executes the commands bound to them, and displays any changes to buffers that result. We call this process the *main loop*. Epsilon loops until you call the `leave_recursion()` primitive, as described on page 285. The steps in the main loop are as follows:

- Epsilon resets the `in_echo_area` variable. See page 236.

- Epsilon calls the `check_abort()` primitive to see if you pressed the abort key since the last time `check_abort()` was called. If so, an abort happens. See page 284.

- Epsilon sets the current buffer to be the buffer connected to the current window.

- Epsilon calls `maybe_refresh()`, so that all windows are brought up to date if the next key is not ready yet.

- Epsilon calls `undo_mainloop()`, to make sure undo information is kept for the current buffer, and to tell the undo system that future buffer changes will be part of the next command.

- Epsilon sets the `this_cmd` and `has_arg` variables to 0, and the `iter` variable to 1. See below.

- Epsilon calls the EEL subroutine `getkey()`. This subroutine in turn calls the `wait_for_key()` primitive to wait for the next key, mouse click, or other event.

- Epsilon executes the new key by calling the primitive `do_topkey()` as described on page 327.

- Epsilon sets the `prev_cmd` variable to the value in `this_cmd`.

```
user short this_cmd;
user short prev_cmd;
invisible_cmd()
```

Some commands behave differently depending on what command preceded them. For example, up-line behaves differently when the previous command was also up-line. To get this behavior, the command acts differently if `prev_cmd` is set to a certain value and sets `this_cmd` to that value itself. Epsilon copies the value in `this_cmd` to `prev_cmd` and then clears `this_cmd`, each time through the main loop.

Sometimes a command doesn't wish to be counted when determining the previous command. For example, when you move the mouse, Epsilon is actually running a command. But the up-line command of the previous example must behave the same, whether or not you happen to move the mouse between one up-line and the next. A command may call the `invisible_cmd()` primitive to make commands like up-line ignore it. (In fact, the primitive simply sets `this_cmd` equal to `prev_cmd`.)

```
user char has_arg;
user int iter;
```

Numeric arguments work using the `has_arg` and `iter` variables. The main loop resets `iter` to 1 and `has_arg` to 0. The argument command sets `iter` to the value of the argument, and sets `has_arg` to 1 so other commands can distinguish an argument of 1 from no argument. The `do_command()` primitive, described on page 327, will repeatedly execute a command while `iter`'s value is greater than one, subtracting one from `iter`'s value with each execution. If a command wants to handle arguments itself, it must set `iter` to one or less before returning, or the main loop will call it again.

```
user short cmd_len;
```

Any command may get more keys using the `wait_for_key()` primitive (usually by calling `getkey()`; see page 302). Epsilon counts the keys used so far by the current command and stores the value in the variable `cmd_len`. This counter is reset to zero each time Epsilon goes through the main loop. The counter doesn't count mouse keys or other events that appear as keys.

8.6.8 Bindings

Epsilon lets each buffer have a different set of key bindings appropriate to editing the type of text in that buffer. For instance, while in a buffer with EEL source code, a certain key could indent the current function. The same key might indent a paragraph in a buffer with text.

A key table stores a set of key bindings. A key table acts like a very large array, with one entry for each key on the keyboard. Each entry in the array contains an index into the name table. (See page 291.) If the value of a particular entry is negative or zero, it means the key is undefined according to that table.

Since the key codes that index a key table can be very large numbers, with big gaps between entries, Epsilon doesn't actually store key tables as arrays, but they act like arrays in EEL. (Internally, Epsilon stores a key table as a type of sparse array.) The `MAXKEYS` macro holds a number bigger than the biggest possible key code.

```
#define key_t        int
set_range(key_t *table, int i, int value, int cnt)
int get_range(key_t *table, int i, int value)
int find_next_entry(key_t *table, int value, int bound)
```

EEL code that wants to perform some operation on groups of keys can't simply use a loop like `for (i = 0; i < MAXKEYS; i++)` to do so; that would be too slow. Instead, there are primitives specifically designed for such purposes.

To set a range of key codes entries to a particular value, use the `set_range()` primitive. It sets the `cnt` entries starting at index `i` in the key table `table` to the specified value. The `key_t` type represents the type of a key table entry.

To find the length of a run of identical values in a key table array, use the `get_range()` primitive. It returns the number of entries, starting at index `i`, with the specified value. For instance, if `k` is a key table, and `k[5]`, `k[6]`, and `k[7]` equal 123, but `k[8]` equals 456, then `get_range(k, 5, 123)` would return 3, and `get_range(k, 8, 123)` would return 0.

The `find_next_entry()` primitive searches for the index of the next member of the key table array `k` with the specified value. It starts looking at position `bound` in the array, skipping past index entries less than or equal to `bound`. It returns −1 if there are no more entries set to the specified value.

The EEL header file `oldkeys.h` provides sample implementations of the above primitives. If the number of possible keys were much smaller, you could use them instead of the above primitives. If you write EEL source code that must also work in older versions of Epsilon, you can include that file. Your EEL source code will use the definitions in `oldkeys.h` when you compile it in old versions of Epsilon; under Epsilon 12 or later, it will use the above primitives.

```
buffer short *mode_keys;
short *root_keys;
keytable reg_tab, c_tab;
```

Epsilon uses two key tables in its search for the binding of a key. First it looks in the key table referenced by the buffer-specific variable `mode_keys`. If the entry for the key is negative, Epsilon considers the command unbound and signals an error. If the entry for the key is 0, as it usually is, Epsilon uses the entry in the key table referenced by the variable `root_keys` instead. If the resulting entry is zero or negative, Epsilon considers the key unbound. If it finds an entry for the key that is a positive number, Epsilon considers that number the key's binding. The number is actually an index into the name table.

Most entries in a key table refer to commands, but an entry may also refer to a subroutine (if it takes no arguments), to a keyboard macro, or to another key table. For example, the entry for Ctrl-X in the default key table refers to a key table named `cx_tab`, which contains the Ctrl-X commands. The entry for the find-file command bound to Ctrl-X Ctrl-F appears in the `cx_tab` key table.

Normally in Epsilon the `root_keys` variable points to the `reg_tab` array. The `mode_keys` variable points to one of the many mode-specific tables, such as `c_tab` for C mode.

```
int new_table(char *name)
int make_anon_keytable()              /* control.e */
short *index_table(int index)
```

Key tables are usually defined with the `keytable` keyword as described on page 182. If a key table's name is not known when the routine is compiled, the `new_table()` primitive can be used. It makes a new key table with the given name. All entries in it are 0.

The `make_anon_keytable()` subroutine defined in control.e calls `new_table()`, first choosing an unused name for the table. The `index_table()` function takes a name table index and retrieves the key table it refers to.

```
fix_key_table(short *ftab, int fval, short *ttab, int tval)
copy_key_table(short *from, short *to)
set_list_keys(short *tab)
```

The `fix_key_table()` subroutine copies selected key table information from one key table to another. For each key in `ftab` bound to the function `fval`, the subroutine binds that key in `ttab` to the function `tval`. The `copy_key_table()` subroutine copies an entire key table. The `set_list_keys()` subroutine sets the 'n' and 'p' keys to move up or down by lines.

```
do_topkey()
run_topkey()
```

When Epsilon is ready to execute a key in its main loop, it calls the primitive `do_topkey()`. This primitive searches the key tables for the command bound to the current key, as described above. When it has found the name table index, it calls `do_command()`, below, to interpret the command.

The `run_topkey()` subroutine provides a wrapper around `do_topkey()` that resets `iter` and similar variables like the main loop does. An EEL subroutine that wants to retrieve keys itself and execute them as if the user typed them at command level can call this subroutine.

```
do_command(int index)
user short last_index;
```

The `do_command()` primitive executes the command or other item with the supplied name table index. If the index is invalid, then the `quick_abort()` primitive is called. Otherwise, the index is copied to the `last_index` variable, so the help system can find the name of the current command (among other uses).

If the name table index refers to a command or subroutine, Epsilon calls the function. When it returns, Epsilon checks the `iter` variable. If it is two or more, Epsilon proceeds to call the same function repeatedly, decrementing `iter` each time, so that it calls the function a total of `iter` times. See page 325.

```
key_t *table_keys;
int table_count;
table_prompt()                    /* control.e */
```

If the entry in the name table that `do_command()` is to execute contains another table, Epsilon gets another key. First, Epsilon updates the primitive array `table_keys`. It contains the prefix keys entered so far in the current command, and `table_count` contains their number. Next, Epsilon calls the EEL subroutine `table_prompt()` if it exists to display a prompt for the new key. The version of this subroutine that's provided with Epsilon uses `mention()`, so the message may not appear immediately. Epsilon then calls the EEL subroutine `getkey()` to read a new key and clears the echo area of the prompt. Epsilon then interprets the key just as the `do_topkey()` primitive would, but using the new key table. If both `mode_keys` and `root_keys` provided a table as the entry for the first key, the values from each are used as the new mode and root key tables.

```
do_again()
```

The `do_again()` primitive reinterprets a key using the same pair of mode and root tables that were used previously. The value in the variable `key` may, of course, be different. Epsilon uses this primitive in commands such as alt-prefix.

Epsilon handles EEL subroutines without parameters in the name table in the same way as commands, as described above. If the entry is for a keyboard macro, the only other legal name table entry, Epsilon goes into a recursive edit level and begins processing the keys in the macro. It saves the macro internally so that future requests for a key will return characters from the macro, as described on page 302. It also saves the value of `iter`, so the macro will iterate properly. When the macro runs out of keys, Epsilon automatically exits the recursive edit level, and returns from the call to `do_again()`. (When `macro-runs-immediately` is nonzero, running a macro doesn't enter a recursive edit level, but returns immediately. Future key requests will still come from the macro until it's exhausted.)

```
short ignore_kbd_macro;
```

Epsilon provides a way for a keyboard macro to suspend itself and get input from the user, then continue. Set the `ignore_kbd_macro` variable nonzero to get keyboard input even when a macro is running. The pause-macro command uses this variable.

```
short *ask_key(char *pr, char *keyname, int ask)
int key_binding[30];        // ask_key() puts key info here
```

The `ask_key()` subroutine defined in basic.e duplicates the logic of the main loop in getting the sequence of keys that make up a command. However, it prompts for the sequence and doesn't run the command at the end. Commands like bind-to-key that ask for a key and accept a sequence of key table keys use it.

The `ask_key()` subroutine returns a pointer to the entry in the key table that was finally reached. The value pointed to is the name table index of the command the key sequence invokes.

This subroutine stores the key sequence in the `keyname` parameter in text form (as "Ctrl-X f", for example). It also copies the key sequence into the global variable `key_binding`. The key sequence is in macro format, so in the example of Ctrl-X f, `key_binding[1]` would hold `CTRL('X')`, `key_binding[2]` would hold `'f'`, and `key_binding[0]` would hold 3, the total number of entries in the array.

If `ask` is nonzero and the user presses a key like ⟨Backspace⟩ with both a generic and a specific interpretation, Epsilon asks the user which one he wants. If `ask` is zero, Epsilon assumes the specific key is meant.

```
full_getkey(char *pr, int code)          /* basic.e */


    /* for full_getkey() */
#define ALTIFY_KEY      1
#define CTRLIFY_KEY     2
```

The `full_getkey()` subroutine defined in basic.e gets a single key from the keyboard, but recognizes the prefix keys ⟨Esc⟩ and Ctrl-^. The `ask_key()` subroutine uses it, as well as the commands bound to the prefix keys above. It takes a prompt to display and a bit pattern (from eel.h) to make it act as if certain of the above keys had already been typed. For example, the ctrl-prefix command calls this subroutine with the value `CTRLIFY_KEY`. It leaves the key that results in the `key` primitive.

```
name_macro(char *name, key_t *keys)
```

Epsilon has no internal mechanism for capturing keyboard keys to build a macro (this is done in the `getkey()` subroutine defined in control.e), but once a macro has been built Epsilon can name it and make it accessible with the `name_macro()` function. It takes the name of the macro to create, and the sequence of keys making up the macro in an array of short ints. This array is in the same format that `get_keycode()` uses. That is, the first element of the array contains the number of valid elements in the array (including the first one). The actual keys in the macro follow. The `name_macro()` primitive makes a copy of the macro it is given, so the array can be reused once the macro has been defined.

```
key_t *get_macro(int index)
```

The `get_macro()` primitive can retrieve the keys in a defined keyboard macro. It takes the name table index of a macro, and returns a pointer to the array containing the macro, in the same format as `name_macro()`.

```
int list_bindings(int start, short *modetable,
                  short *roottable, int find)
```

The `list_bindings()` primitive quickly steps through a pair of key tables, looking for entries that have a certain name table index. It takes mode and root key tables, the name table index to find, and either -1 to start at the beginning of the key tables, or the value it returned on a previous call. It returns the index into the key table, or -1 if there are no more matching entries. For each position in the tables, Epsilon looks at the value in the mode key table, unless it is zero. In that case, it uses the root table.

In addition to the matches, `list_bindings()` also stops on each name table index corresponding to a key table, since these must normally be searched also. For example, the following file defines a command that counts the number of separate bindings of any command.

```
#include "eel.h"

command count_bindings()
{
    char cmd[80];

    get_cmd(cmd, "Count bindings of command", "");
    if (*cmd)
        say("The %s command has %d bindings", cmd,
            find_some(mode_keys,
                      root_keys, find_index(cmd)));
}

        /* count bindings to index in table */
int find_some(modetable, roottable, index)
        short *modetable, *roottable;
{
    int i, total = 0, found;

    i = list_bindings(-1, modetable, roottable, index);
    while (i != -1) {

        found = (modetable[i]
                        ? modetable[i] : roottable[i]);
```

```
            if (found == index)
                total++;
            else
                total += find_some(index_table(found),
                             index_table(found), index);

            i = list_bindings(i, modetable, roottable, index);
        }
        return total;
    }
```

8.7 Defining Language Modes

There are several things to be done to define a new mode. Suppose you wish to define a mode called
reverse-mode in which typing letters inserts them backwards, so typing "abc" produces "cba", and yanking
characters from a kill buffer inserts them in reverse order. First, define a key table for the mode with the
keytable keyword, and put the special definitions for that mode in the table:

```
    keytable rev_tab;

    command reversed_normal_character()
    {
        normal_character();
        point--;
    }

    when_loading()
    {
        int i;

        for (i = 'a'; i <= 'z'; i++)
            rev_tab[toupper(i)] = rev_tab[i] = (short)
                            reversed_normal_character;
    }


    command yank_reversed() on rev_tab[CTRL('Y')]
    {
            ...
    }
```

Now define a command whose name is that of the mode. It should set mode_keys to the new table and
major_mode to the name of the mode, and then call the subroutine make_mode() to update the mode line:

```
    command reverse_mode()
    {
        mode_keys = rev_tab;           /* use these keys */
        major_mode = strsave("Reverse");
        make_mode();
    }
```

Instead of using `strsave()`, a better approach is to define a character array variable with the mode name, and set `major_mode` to that. The mode name in `major_mode`, with the addition of "`-mode`", should be the name of a command that goes into that mode.

If you want Epsilon to go into that mode automatically when you find a file with the extension .rev (as it goes into C mode with .c files, for instance), define a function named `suffix_rev()` which calls `reverse_mode()`. The EEL subroutine `find_it()` defined in files.e automatically calls a function named `suffix_`*ext* (where *ext* is the file's extension) whenever you find a file, if a function with that name exists. It tries to call the `suffix_none()` function if the file has no suffix. If it can't find a function with the correct suffix, it will try to call the `suffix_default()` function instead.

```
suffix_rev()
{
    reverse_mode();
}
```

Language modes may wish to define a compilation command. This tells the compile-buffer command on Alt-F3 how to compile the current buffer. For example, `compile_asm_cmd` is defined as `ml "%r"`. (Note that `"` characters must be quoted with `\` in strings.) Use one of the % sequences shown on page 99 in the command to indicate where the file name goes, typically `%f` or `%r`.

The mode can define coloring rules. See page 243 for details. Often, you can copy existing syntax coloring routines like those for .asm or .html files and modify them. They typically consist of a loop that searches for the next "interesting" construct (like a comment or keyword), followed by a `switch` statement that provides the coloring rule for each construct that could be found. Usually, finding an identifier calls a subroutine that does some additional processing (determining if the identifier is a keyword, for instance).

A language mode should set comment variables like `comment-start`. This tells the commenting commands (see page 82) how to search for and create legal comments in the language.

The comment commands look for comments using regular expression patterns contained in the buffer-specific variables `comment-pattern` (which should match the whole comment) and `comment-start` (which should match the sequence that begins a comment, like '/*'). When creating a comment, comment commands insert the contents of the buffer-specific variables `comment-begin` and `comment-end` around the new comment.

SHOWING MATCHING DELIMITERS.

Commands like forward-level that move forward and backward over matching delimiters will (by default) recognize (, [, and { delimiters. It won't know how to skip delimiters inside quoted strings, or similar language-specific features. A language mode can define a replacement delimiter movement function. See page 209 for details.

To let Epsilon automatically highlight matching delimiters in the language when the cursor appears on them, a language mode uses code like this:

```
if (auto_show_asm_delimiters)
    auto_show_matching_characters = asm_auto_show_delim_chars;
```

where references to "asm" are of course replaced by the mode's name. The language mode should define the two variables referenced above:

```
user char auto_show_asm_delimiters = 1;
user char asm_auto_show_delim_chars[20] = "{[]}";
```

The list of delimiters should contain an even number of characters, with all left delimiters in the left half and right delimiters in the right half. (A delimiter that's legal on the left or right should appear in both halves; then the language must provide a `mode_move_level` definition that can determine the proper search direction itself. See page 209.)

Sometimes a mode may wish to highlight delimiters more complicated than single characters, such as BEGIN and END keywords. To do this, the mode should define a function such as `mymode_auto_show_delimiter()` and then set the buffer-specific function pointer variable `mode_auto_show_delimiter` to point to it in that buffer.

Epsilon will then call that function when idle to highlight delimiters. It should return 0 if no highlighting should be done, 1 to make Epsilon try to use the `auto_show_matching_characters` setting described above for simple highlighting, 2 to indicate mismatched delimiters, or 3 to indicate matched delimiters. In the latter two cases it should also display the highlighting, by setting two arrays to mark the appropriate buffer regions, as shown in the example. This sample only demonstrates how to control the highlighting; a typical mode would use smarter rules for finding the matching keywords (ignoring nested pairs, skipping over keywords in comments or strings, and so forth).

```
#include "eel.h"
#include "colcode.h"

int mymode_auto_show_delimiter()
{
    save_var point, case_fold = 1;
    save_var matchstart, matchend, abort_searching = 0;
    init_auto_show_delimiter();   // Must do this first.
    point -= parse_string(-1, "[a-z0-9_]+");
    *highlight_area_start[0] = point;
    if (parse_string(1, "</word>begin</word>")) {
        *highlight_area_end[0] = matchend;
        if (!re_search(1, "</word>end</word>"))
            return 2;
    } else if (parse_string(1, "</word>end</word>")) {
        *highlight_area_end[0] = matchend;
        if (!re_search(-1, "</word>begin</word>"))
            return 2;
    } else
        return 1;
    *highlight_area_start[1] = matchstart;   // Mark the far end.
    *highlight_area_end[1] = matchend;
    modify_region(SHOW_MATCHING_REGION, MRTYPE, REGNORM);
            // Make the highlighting visible.
    return 3;
}
```

Figure 8.3: Highlighting keyword delimiters.

A language mode may also want to set things up so typing a closing delimiter momentarily moves the cursor back to show its matching pair. Binding keys like] and) to the command show-matching-delimiter will accomplish this.

DISPLAYING THE CURRENT FUNCTION'S NAME.

```
c_func_name_finder()  // Sample C mode func name finder.
char display_func_name[];
char must_find_func_name;
int start_of_function;
set_display_func_name()
get_func_name(int idle)
```

A language mode may want to arrange for the name of the current function or similar to appear in the mode line, subject to the `display-definition` variable.

To do this, it must define a function named *modename*`_func_name_finder`, where *modename* is the mode's name as recorded in the `major_mode` variable. The function should write the current function's name to the `display_func_name` variable and return 1, also setting the `start_of_function` variable to a buffer position representing the start of the function, if it can. If point is not in a function, set `display_func_name` to an empty string and return 1.

Epsilon normally runs this function during idle time. If the user presses a key during this function, and the `must_find_func_name` variable is zero, it should stop any slow parsing if it can and return 0.

There are two subroutines that take advantage of such functions. The `set_display_func_name()` subroutine is what Epsilon calls when idle, to update the displayed function name.

The `get_func_name()` subroutine allows EEL code to take advantage of a mode's function name finder at other times. Pass a nonzero value for `idle` if you want it to give up and return should the user press a key. It returns 0 if Epsilon's idea of the function name in `display_func_name` was already up to date, 1 if it wasn't, but it now is, 2 if it couldn't be computed, or 3 if the function gave up to handle a waiting key. Both these functions set `start_of_function` to the start of the current function in this buffer if they can, or -1 otherwise.

H**ELPFUL SUBROUTINES.**

Some subroutines help with mode-specific tasks.

```
int call_by_suffix(char *file, char *pattern)
int get_mode_variable(char *pat)
char *get_mode_string_variable(char *pat)
int get_mode_based_index(char *pat)
```

The `call_by_suffix()` subroutine constructs a function name based on the extension of a given file (typically the file associated with the current buffer). It takes the file name, and a function name with `%s` where the extension (without its leading ".") should be. For example, `call_by_suffix("file.cpp", "tag-suffix-%s")` looks for a subroutine named `tag-suffix-cpp`. (If the given file has no extension, the subroutine pretends the extension was "none".)

If there's no subroutine with the appropriate name, `call_by_suffix()` then replaces the `%s` with "default" and tries to call that function instead. The `call_by_suffix()` subroutine returns 1 if it found some function to call, or 0 if it couldn't locate any suitable function.

The `get_mode_variable()` subroutine searches for a function or variable with a name based on the current mode. Its parameter `pat` must be a printf-style format string, with a `%s` where the current mode's name should appear. The subroutine will look for a function or variable with the resulting name. A variable by that name must be numeric; the subroutine will return its value. A function by that name must take no parameters and return a number; this subroutine will call it and return its value. In either case it will set the

`got_bad_number` variable to zero. If `get_mode_variable()` can't locate a suitable function or variable, it sets `got_bad_number` nonzero.

The `get_mode_string_variable()` subroutine retrieves the value of a string variable whose name depends on the current mode. The name may also refer to a function; its value will be returned. It constructs the name by using `sprintf()`; `pat` should contain a `%s` and no other `%` characters; the current mode's name will replace the `%s`. If there's no such variable or function with that name, it returns NULL. The subroutine sets the `got_bad_number` variable nonzero to indicate that there was no such name, or zero otherwise.

The `get_mode_based_index()` subroutine looks for a name table entry of any sort (a function, variable, key table, etc.) with a name built by replacing the `%s` sequence in the specified pattern with the name of the current mode. If there is none, it substitutes `"default"` for the mode name and tries again. It returns the name table index of the entry it found, or zero if none.

```
int guess_mode_without_extension(char *res, char *pat)
```

The `guess_mode_without_extension()` subroutine tries to determine the correct mode for a file without an extension, mostly by examining its text. It can detect some Perl and C++ header files that lack any .perl or .hpp extension, as well as makefiles (based simply on the file's name) and various other sorts of files. If it can determine the mode, it uses `pat` as a pattern for `sprintf()` (so it should contain one `%s` and no other `%`'s) and sets `res` to the `pat`, with its `%s` replaced by the mode name. Then it returns 1. If it can't guess the mode it returns 0.

```
mode_default_settings()
```

The `mode_default_settings()` subroutine resets a number of mode-specific variables to default settings. A command that establishes a mode can call this subroutine, if it doesn't want to provide explicit settings for all the usual mode-specific variables, such as comment pattern variables.

```
zeroed buffer (*buffer_maybe_break_line)();
```

The auto-fill minor mode normally calls a function named `maybe_break_this_line()` to break lines. A major mode may set the buffer-specific function pointer `buffer_maybe_break_line` to point to a different function; then auto-fill mode will call that function instead, for possibly breaking lines as well as for turning auto-fill on or off, or testing its state.

A `buffer_maybe_break_line` function will be called with one numeric parameter. If 0 or 1, it's being told to turn auto-fill off or on. The function may interpret this request to apply only to the current buffer, or to all buffers in that mode. It should return 0.

If its parameter is 2, it's being asked whether auto-fill mode is on. It should return a nonzero value to indicate that auto-fill mode is on.

If its parameter is 3, it's being asked to perform an auto-fill, if appropriate, triggered by the key in the variable `key`, which has not yet been inserted in the buffer. It may simply return 1 if the line is not wide enough yet, or after it has broken the line. Epsilon will then insert the key that triggered the filling request. If it returns zero, Epsilon will skip inserting the key that triggered the filling.

8.7.1 Language-specific Subroutines

```
int find_c_func_info(char *type, char *class,
                     char *func, int stop_on_key)
```

The `find_c_func_info()` subroutine gets info on the function or class defined at point in the current C-mode buffer, by parsing the buffer. It sets `class` to the class name of the current item, if any, and `func` to the function name if any. It sets `type` to `"class"`, `"struct"`, or `"union"` if it can determine which is appropriate. Outside a function or class definition, the above will be set to `""`. You may pass NULL for any of the above parameters if you don't need that information.

If `stop_on_key` is nonzero, and the user presses a key while the function is running, the function will immediately return `-1` without setting the above variables. Otherwise the function returns a bit pattern: `CF_INFO_TYPE` if `type` was set non-empty; `CF_INFO_CLASS` if `class` was set non-empty; and `CF_INFO_FUNC` if `func` was set non-empty. In addition to zero, only these combination can occur:

CF_INFO_TYPE	CF_INFO_CLASS	CF_INFO_FUNC
*	*	
		*
	*	*
*	*	*

Chapter 9

Error Messages

This chapter lists some of the error messages Epsilon can produce, with explanations. In general, any error numbers produced with error messages are returned from the operating system.

Argument list mismatch in call. An EEL function was called with the wrong number of parameters. Perhaps you tried to call an EEL function by name, from the command line. Only functions that take no formal parameters can be called this way.

Can't find tutorial. Install first. Epsilon tried to load its tutorial file, since you started it with the -teach option, but can't find it. The tutorial is a file named eteach, located in Epsilon's main directory.

Can't interpret type of *variable-name***.** You can only set or show variables that have numbers or characters in them.

COMSPEC missing from environment. Epsilon needs a valid COMSPEC environment variable in order to run another program. See page 9.

Couldn't exec: error *number***.** You tried to run a program from within Epsilon, and Epsilon encountered an error trying to invoke that program. The *number* denotes the error code returned by the operating system. Also see the previous error.

Debug: can't read source file *filename***.** Epsilon's EEL debugger tried to read an EEL source file, but couldn't find it. Epsilon gets a source file's pathname from the EEL compiler's command line. If you compiled an EEL file with the command "eel dir/file.e", Epsilon will look for a file named "dir/file.e". Check that your current directory is the same as when you ran the EEL compiler.

Don't know how to tag the file *filename***.** Epsilon only knows how to tag files with certain extensions like .c, .h, .e, and .asm. Using EEL, you can tell Epsilon how to tag other types of files, though. See page 270.

Files not deleted. An error occurred when the dired command tried to delete the file or directory. You can only delete empty directories.

Invalid or outdated byte code file *filename***.** The byte code file Epsilon tried to load was created with another version of Epsilon, was empty, or was illegal in some other way. Try compiling it again with the EEL compiler.

filename **is not a directory.** You specified *filename* in an -fs flag, telling Epsilon to create its temporary files there, but it isn't a directory.

Macro definition buffer full: keyboard macro defined. You tried to define a macro of more than 500 keys from the keyboard. This might happen because you forgot to close a macro definition with the Ctrl-X) command. If you really want to define such a big macro, use the command file mechanism (see page 131) or change the MAX_MACRO constant defined in eel.h and recompile control.e using EEL.

Macro nesting too deep. All macros canceled. An Epsilon keyboard macro can call another keyboard macro recursively (but only if the calling macro is defined by a command file—see page 125). To catch runaway recursive macros, Epsilon puts a limit on the depth of keyboard macro recursion. Epsilon allows unlimited *tail-recursion*: if a macro calls another macro with its last keystrokes, Epsilon finishes the original macro call before beginning the next one.

Only one window. The diff and compare-windows commands compare the current window with the next window on the screen, but there's only one window.

function **undefined or of wrong type.** Epsilon initialized itself exclusively from a bytecode file (without reading a state file), since you gave the -b flag, but that file didn't define a function or variable that Epsilon needs to run. See page 299. To load a bytecode file, in addition to Epsilon's usual commands, use the -l flag, not the -b flag.

Appendix A

Index

C

N

O

T