GNU EMACS MANUAL

By Richard M. Stallman

GNU Emacs Manual

First Edition, Emacs Version 16

June 1985

Richard Stallman

Summary Table of Contents

Table of Contents

Preface

This manual documents the use and simple customization of the Emacs editor. The reader is not expected to be a programmer. Even simple customizations do not require programming skill, but the user who is not interested in customizing can ignore the scattered customization hints.

This is primarily a reference manual, but can also be used as a primer. However, I recommend that the newcomer first use the on-line, learn-by-doing tutorial, which you get by running Emacs and typing `C-h t`. With it, you learn Emacs by using Emacs on a specially designed file which describes commands, tells you when to try them, and then explains the results you see. This gives a more vivid introduction than a printed manual.

On first reading, you need not make any attempt to memorize chapters one and two, which describe the notational conventions of the manual and the general appearance of the Emacs display screen. It is enough to be aware of what questions are answered in these chapters, so you can refer back when you later become interested in the answers. After reading chapter four you should practice the commands there. The next few chapters describe fundamental techniques and concepts that are referred to again and again. It is best to understand them thoroughly, experimenting with them if necessary.

To find the documentation on a particular command, look in the index. Keys (character commands) and command names have separate indexes just for them. There is also a glossary, with a cross reference for each term.

GNU Emacs is a member of the Emacs editor family. There are many Emacs editors, all sharing common principles of organization. For information on the underlying philosophy of Emacs and the lessons learned from its development, write for a copy of AI memo 519a, "Emacs, the Extensible, Customizable Self-Documenting Display Editor", to

Publications Department
Artificial Intelligence Lab
545 Tech Square
Cambridge, MA 02139

Introduction

You are about to read about GNU Emacs, the Unix/GNU incarnation of the advanced, self-documenting, customizable, extensible real-time display editor Emacs.

We say that Emacs is a *display* editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands. See chapter 1 [Screen], page 5.

We call it a *real-time* editor because the display is updated very frequently, usually after each character or pair of characters you type. This minimizes the amount of information you must keep in your head as you edit. See chapter 4 [Basic Editing], page 17.

We call Ema‑s advanced because it provides facilities that go beyond simple insertion and deletion: filling of text; automatic indentation of programs; viewing two or more files at once; and dealing in terms of characters, words, lines, sentences, paragraphs, and pages, as well as expressions and comments in several different programming languages. It is much easier to type one command meaning "go to the end of the paragraph" than to find that spot with simple cursor keys.

Self-documenting means that at any time you can type a special character, `Control-h`, to find out what your options are. You can also use it to find out what any command does, or to find all the commands that pertain to a topic. See chapter 8 [Help], page 33.

Customizable means that you can change the definitions of Emacs commands in little ways. For example, if you use a programming language in which comments start with '`<**`' and end with '`**>`', you can tell the Emacs comment manipulation commands to use those strings. Another sort of customization is rearrangement of the command set. For example, if you prefer the four basic cursor motion commands (up, down, left and right) on keys in a diamond pattern on the keyboard, you can have it. See chapter 29 [Customization], page 159.

Extensible means that you can go beyond simple customization and write entirely new commands, programs in the Lisp language. Emacs is an "on-line extensible" system, which means that it is divided into many functions that call each other, any of which can be redefined in the middle of an editing session. Any part of Emacs can be replaced without making a separate copy of all of Emacs. Most of the editing commands of Emacs are written in Lisp already; the few exceptions could have been written in Lisp but are written in C for efficiency. Although only a programmer can write an extension, anybody can use it afterward.

1. The Organization of the Screen

Emacs divides the screen into several areas, each of which contains its own sorts of information. The biggest area, of course, is the one in which you usually see the text you are editing.

When you are using Emacs, the screen is divided into a number of *windows*. Initially there is one text window occupying all but the last line, plus the special *echo area* or *minibuffer window* in the last line. The text window can be subdivided horizontally or vertically into multiple text windows, each of which can be used for a different file (see chapter 19 [Windows], page 91). The window that the cursor is in is the *selected window*, in which editing takes place. The other windows are just for reference unless you select one of them.

Each text window's last line is a *mode line* which describes what is going on in that window. It is in inverse video if the terminal supports that, and contains text that starts like '`-----Emacs:` *something*'. Its purpose is to indicate what buffer is being displayed in the window above it; what major and minor modes are in use; and whether the buffer's text has been changed.

1.1 Point

When Emacs is running, the terminal's cursor shows the location at which editing commands will take effect. This location is called *point*. Other commands move point through the text, so that you can edit at different places in it.

While the cursor appears to point *at* a character, point should be thought of as *between* two characters; it points *before* the character that the cursor appears on top of. Sometimes people speak of "the cursor" when they mean "point", or speak of commands that move point as "cursor motion" commands.

Terminals have only one cursor, and when output is in progress it must appear where the typing is being done. This does not mean that point is moving. It is only that Emacs has no way to show you the location of point except when the terminal is idle.

Each Emacs buffer has its own point location. A buffer that is not being displayed remembers where point is so that it can be seen when you look at that buffer again.

When there are multiple text windows, each window has its own point location. The cursor shows the location of point in the selected window. This also is how you can tell which window is selected. If the same buffer appears in more than one window, point can be moved in each window independently.

Point is called *dot* in the Emacs source code and on-line documentation. Both names come from the character '`.`', which was the command in TECO (the language in which the original Emacs was written) for accessing the value now called 'point'. The name 'point' is preferred and 'dot' will being phased out.

1.2 The Echo Area

The line at the bottom of the screen (below the mode line) is the *echo area*. It is used to display small amounts of text for several purposes.

Echoing means printing out the characters that you type. Emacs does not echo single-character keys, and does not echo any keys if you type the characters with no long pause, but if you pause for more than a second in the middle of a multi-character key, then all the characters typed so far are echoed. This is intended to *prompt* you for the rest of the key. Once the beginning of a key has been echoed, all the rest is echoed as soon as it is typed; so either the entire key or none of it is echoed. This behavior is designed to give confident users fast response, while giving hesitant users maximum feedback.

If a command cannot be executed, it may print an *error message* in the echo area. Error messages are accompanied by a beep or by flashing the screen. Also, any input you have typed ahead is thrown away when an error happens.

Some commands print informative messages in the echo area. These messages look much like error messages, but they are not announced with a beep and do not throw away input. Sometimes the message tells you what the command has done, when it is not obvious from looking at the text being edited. Sometimes the sole purpose of a command is to print a message giving you specific information. For example, the command `C-x =` is used to print a message describing the character position of point in the text and its current column in the window. Commands that take a long time often display messages ending in '...' while they are working, and add 'done' at the end.

The echo area is also used to display the *minibuffer*, a window that is used for reading arguments to commands, such as the name of a file to be edited. When the minibuffer is in use, the echo area begins with a prompt string that ends with a colon; also, the cursor appears in that line because it is the selected window. You can always get out of the minibuffer by typing `C-g`. See chapter 6 [Minibuffer], page 25.

1.3 The Mode Line

Each text window's last line is a *mode line* which describes what is going on in that window. When there is only one text window, the mode line appears right above the echo area. The mode line is in inverse video if the terminal supports that, starts and ends with dashes, and contains text like '`Emacs:` *something*'.

If a mode line has something else in place of '`Emacs:` *something*', then the window above it is in a special subsystem such as Rmail. The mode line then indicates the status of the subsystem.

Normally, the mode line has the following appearance:

---ch"-Emacs: buf (major minor)—-pos%——

This serves to indicate various information about the buffer being displayed in the window: the buffer's name, what major and minor modes are in use, whether the buffer's text has been changed, and how far down the buffer you are currently looking. The top level mode line has this format:

ch contains two stars '`**`' if the text in the buffer has been edited (the buffer is"modified"), or '`--`' if the buffer has not been edited. Exception: for a read-only buffer, it is '`%%`'.

buf is the name of the window's chosen *buffer*. The chosen buffer in the selected window (the window that the cursor is in) is also Emacs's selected buffer, the one that editing takes place in. When we speak of what some command does to "the buffer", we are talking about the currently selected buffer. See chapter 18 [Buffers], page 85.

major is the name of the *major mode* in effect in the buffer. At any time, each buffer is in one and only one of its possible major modes. The major modes available include Fundamental mode (the least specialized), Text mode, Lisp mode, C mode, and others. See chapter 20 [Major Modes], page 95, for details of how the modes differ and how to select one.

minor is a list of some of the *minor modes* that are turned on at the moment in the window's chosen buffer. '`Fill`' means that Auto Fill mode is on. '`Abbrev`' means that Word Abbrev mode is on. '`Ovwrt`' means that Overwrite mode is on. See section 29.1 [Minor Modes], page 159, for more information. '`Narrow`' means that the buffer being displayed has editing restricted to only a portion of its text. This is not really a minor mode, but is like one. See chapter 28 [Narrowing], page 153.

pos tells you whether there is additional text above the top of the screen, or below the bottom. If your file is small and it is all on the screen, *pos* is 'All' is omitted. Otherwise, it is 'Top' if you are looking at the beginning of the file, 'Bot' if you are looking at the end of the file, or '*nn*%', where *nn* is the percentage of the file above the top of the screen.

Some other information about the state of Emacs can also be displayed among the minor modes. 'Def' means that a keyboard macro is being defined; although this is not exactly a minor mode, it is still useful to be reminded about. See section 29.3 [Keyboard Macros], page 164.

In addition, if Emacs is currently inside a recursive editing level, square brackets ('[...]') appear around the parentheses that surround the modes. If Emacs is in one recursive editing level within another, double square brackets appear, and so on. Since this information pertains to Emacs in general and not to any one buffer, the square brackets appear in every mode line on the screen or not in any of them. See section 27.1 [Recursive Edit], page 149.

1.4 Variables Controlling Display

This section contains information for customization only. Beginning users should skip it.

The variable mode-line-inverse-video controls whether the mode line is displayed in inverse video (assuming the terminal supports it); nil means don't do so.

If the variable inverse-video is non-nil, Emacs attempts to invert all the lines of the display from what they normally are.

If the variable visible-bell is non-nil, Emacs attempts to make the whole screen blink when it would normally make an audible bell sound. This variable has no effect if your terminal does not have a way to make the screen blink.

The variable echo-keystrokes controls the echoing of multi-character keys; its value is the number of seconds of pause required to cause echoing to start, or zero meaning don't echo at all.

If the variable ctl-arrow is nil, control characters also are displayed with octal escape sequences, all except RET and TAB. This variable has a separate value in each buffer; in new buffers, its value is initialized from the variable default-ctl-arrow.

Normally, a tab character in the buffer is displayed as whitespace which extends to the next display tab stop position, and display tab stops come at intervals equal to eight spaces. The number of spaces per tab is controlled by the variable **tab-width**, which is local to every buffer just like **ctl-arrow** and gets its value in a new buffer from **default-tab-width**. Note that how the character tab in the buffer is displayed has nothing to do with the definition of **TAB** as a command.

2. Characters, Keys and Commands

This chapter explains the character set used by Emacs for input commands and for the contents of files, and also explains the concepts of *keys* and *commands* which are necessary for understanding how your keyboard input is understood by Emacs.

2.1 The Emacs Character Set

GNU Emacs uses the ASCII character set, which defines 128 different character codes. Some of these codes are assigned graphic symbols such '**a**' and '**=**'; the rest are control characters, such as `Control-a` (also called `C-a` for short). `C-a` gets its name from the fact that you type it by holding down the `CTRL` key and then pressing **a**. There is no distinction between `C-a` and `C-A`; they are the same character.

Some control characters have special names, and special keys you can type them with: `RET`, `TAB`, `LFD`, `DEL` and `ESC`. The space character is usually referred to below as `SPC`, even though strictly speaking it is a graphic character whose graphic happens to be blank.

Emacs extends the 7-bit ASCII code to an 8-bit code by adding an extra bit to each character. This makes 256 possible command characters. The additional bit is called Meta. Any ASCII character can be made Meta; Meta characters include `Meta-a` (`M-a`, for short), `M-A` (not the same character as `M-a`, but those two characters normally have the same meaning in Emacs), `M-RET`, and `M-C-a`.

Some terminals have a `META` key, and allow you to type Meta characters by holding this key down. Thus, `Meta-a` is typed by holding down `META` and pressing a. Such a key is not always labeled `META`, however, as it is usually a special option from the manufacturer. If there is no `META` key, you can still type Meta characters using two-character sequences starting with `ESC`. Thus, to enter `M-a`, you could type `ESC a`. This is allowed on terminals with Meta keys, too, in case you have formed a habit of doing it.

Emacs believes the terminal has a `META` key if the variable `meta-flag` is non-`nil`. Normally this is set automatically according to the termcap entry for your terminal type. However, sometimes the termcap entry is wrong, and then it is useful to set this variable yourself.

Emacs buffers also use an 8-bit character set, because bytes have 8 bits, but only the ASCII characters are considered meaningful. ASCII graphic characters in Emacs buffers are displayed with their graphics. LFD is the same as a newline character; it is displayed by starting a new line. TAB is displayed by moving to the next tab stop column (usually every 8 columns). Other control characters are displayed as a caret ('^') followed by the non-control version of the character; thus, C-a is displayed as '^A'. Non-ASCII characters 128 and up are displayed with octal escape sequences; thus, character code 243 (octal), also called M-# when used as an input character, is displayed as '\243'.

2.2 Keys

A *key*—short for *key sequence*—is a sequence of characters that is all part of specifying a single Emacs command to be run. If the characters are enough to specify a command, they form a *complete key*.

A single character is always a key; whether it is complete depends on its meaning in Emacs. Most single characters are complete Emacs commands. C-h, C-x and ESC are the only ones that are not complete.

A sequence of characters that is not enough to specify an Emacs command is called a *prefix key*. A prefix key is the beginning of a series of longer sequences that are valid keys; adding any single character to the end of the prefix gives a valid key, which could be defined as an Emacs command. For example, C-x is normally defined as a prefix, so C-x and the next input character combine to make a two-character key. There are 256 different two-character keys starting with C-x, one for each possible second character. Most of these two-character keys starting with C-x are standardly defined as Emacs commands. The most notable ones are C-x C-f or C-x s (see chapter 16 [Files], page 71).

Adding one character to a prefix key does not have to form a complete key. It could make another, longer prefix. For example, C-x 4 is itself a prefix that leads to 256 different three-character keys, including C-x 4 f, C-x 4 b and so on. It would be possible to define one of those three-character sequences as a prefix, creating a series of four-character keys, but we did not define any of them this way.

All told, the prefix keys in Emacs are C-x, C-h, C-x 4, and ESC. C-c is an additional prefix but only in certain modes.

2.3 Keys and Commands

This manual is full of passages that tell you what particular keys do. But Emacs does not assign meanings to keys directly. Instead, Emacs assigns meanings to *functions*, and then gives keys their meanings by *binding* them to functions.

A function is a Lisp object that can be executed as a program. Usually it is a Lisp symbol which has been given a function definition; every symbol has a name, usually made of a few English words separated by dashes, such as next-line or forward-word. It also has a *definition* which is a Lisp program; this is what makes the function do what it does. Only some functions can be the bindings of keys; these are functions whose definitions use interactive to specify how to call them interactively. Such functions are called *commands*, and the name of a symbol that is a command is called a *command name*. More information on this subject will appear in the *GNU Emacs Lisp Manual* (which is not yet written).

The bindings between keys and functions are recorded in various tables called *keymaps*. See section 29.4 [Keymaps], page 166.

When we say that "`C-n` moves down vertically one line" we are glossing over a distinction that is irrelevant in ordinary use but is vital in understanding how to customize Emacs. It is the function next-line that is programmed to move down vertically. `C-n` has this effect *because* it is bound to that function. If you rebind `C-n` to the function forward-word then `C-n` will move forward by words instead. Rebinding command characters is a common method of customization.

In the rest of this manual, we usually ignore this subtlety to keep things simple. To give the customizer the information he needs, we state the name of the command which really does the work in parentheses after mentioning the key that runs it. For example, we will say that "The command `C-n` (next-line) moves point vertically down," meaning that next-line is a command that moves vertically down and `C-n` is a key that is standardly bound to it.

While we are on the subject of customization information which you should not be frightened of, it's a good time to tell you about *variables*. Often the description of a command will say, "To change this, set the variable mumble-foo." A variable is a name used to remember a value. Most of the variables documented in this manual exist just to permit customization: the variable's value is examined by some command, and changing the value makes the command behave differently. Until you are interested in customizing, you can ignore this information. When you are ready to be interested, readthe basic information on variables, and then the information on individual variables will make sense. See section 29.2 [Variables], page 159.

3. Entering and Exiting Emacs

The simplest way to invoke Emacs is just to type `emacs RET` at the shell.

It is also possible to specify files to be visited, Lisp files to be loaded, and functions to be called, using by giving Emacs arguments in the shell command line. Here are the arguments allowed:

`-q`	Do not load your Emacs init file ~/.emacs.
`-u` *user*	Load *user*'s Emacs init file ~*user*/.emacs instead of your own.
`-t` *device*	Use *device* as the terminal for editing input and output. '`-t`' is recognized only if it is the first command argument.
`-batch`	Run Emacs in *batch mode*, which means that the text being edited is not displayed and the standard Unix interrupt characters `C-z`, `C-c` and `C-\` continue to have their normal effect. Emacs in batch mode outputs to stdout only what would normally be printed in the echo area under program control. Batch mode is used for running programs written in Emacs Lisp from shell scripts, makefiles, and so on. '`-batch`' is recognized only if it is the first command argument.
`-l` *file*	Load a file *file* of Lisp code with load. See section 24.3 [Lisp Libraries], page 132.
`-f` *function*	Call Lisp function *function* with no arguments.
`-kill`	Exit from Emacs without asking for confirmation.
file	Visit *file* using find-file. See section 16.2 [Visiting], page 72.
+*linenum file*	Visit *file* using find-file, then go to line number *linenum* in it.

The command arguments are processed in the order they appear in the command argument list; however, certain switches must be at the front of the list ('`-t`' or '`-batch`') if they are used.

One way to use this is to visit many files automatically:

```
emacs *.c
```

which passes each .c file as a separate argument to Emacs, so that Emacs visits each one.

Here is an advanced example that assumes you have a Lisp program file called hack-c-program.el which, when loaded, performs some useful operation on current buffer, expected to be a C program.

```
emacs -batch foo.c -l hack-c-program -f save-buffer -kill-emacs > log
```

Here Emacs is told to visit foo.c, load hack-c-program.el (which makes changes in the visited file), save foo.c (note that save-buffer is the function that `C-x C-s` is bound to), and then exit to the shell that this command was done with. '`-batch`' guarantees there will be no problem redirecting output to log, because Emacs will not assume that it has a display terminal to work with.

3.1 Exiting Emacs

There are ⁺wo commands for exiting Emacs because there are two kinds of exiting: *suspending* Emacs and *killing* Emacs. *Suspending* means stopping Emacs temporarily and returning control to its superior (usually the shell), allowing you to resume editing later in the same Emacs job, with the same files, same kill ring, same undo history, and so on. This is the usual way to exit. *Killing* Emacs means destroying the Emacs job. You can run Emacs again after killing it, but you will get a fresh Emacs; there is no way to resume the same editing session after it has been killed.

To suspend Emacs, type `C-z` (suspend-emacs).

To kill Emacs, type `C-x C-c` (save-buffers-kill-emacs). A two-character key is used for this to make it harder to type. Unless a numeric argument is used, this command first offers to save any modified buffers. If you do not save them all, it asks for reconfirmation with 'yes' before killing Emacs, since any changes not saved before that will be lost forever. Also, if any subprocesses are still running, `C-x C-c` asks for confirmation about them, since killing Emacs will kill the subprocesses immediately.

In most Unix programs, **but not in Emacs**, the characters `C-z` and `C-c` instantly suspend or kill, respectively. The meanings of `C-z` and `C-x C-c` as keys in Emacs were inspired by the standard Unix meanings of `C-z` and `C-c`, but there is no causal connection. The standard Unix handling of `C-z` and `C-c` are turned off in Emacs. You could customize these keys to do anything (see section 29.4 [Keymaps], page 166).

4. Basic Editing Commands

We now give the basics of how to enter text, make corrections, and save the text in a file. If this material is new to you, you might learn it more easily by running the Emacs learn-by-doing tutorial. To do this, type `Control-h t` (help-with-tutorial).

4.1 Inserting Text

To insert printing characters into the text you are editing, just type them. Except in special modes, Emacs defines each printing character as a key to run the command self-insert, which inserts the character that you typed to invoke it into the buffer at the cursor (that is, at *point*; see section 1.1 [Point], page 5). The cursor moves forward. Any characters after the cursor move forward too. If the text in the buffer is '`FOOBAR`', with the cursor before the '`B`', then if you type `XX`, you get '`FOOXXBAR`', with the cursor still before the '`B`'.

To *delete* text you have just inserted, you can use `DEL` (which runs the command named delete-backward-char). `DEL` deletes the character *before* the cursor (not the one that the cursor is on top of or under; that is the character *after* the cursor). The cursor and all characters after it move backwards. Therefore, if you type a printing character and then type `DEL`, they cancel out.

To end a line and start typing a new one, type `RET` (running the command newline). `RET` operates by inserting a newline character in the buffer. If point is in the middle of a line, `RET` splits the line. Typing `DEL` when the cursor is at the beginning of a line rubs out the newline before the line, thus joining the line with the preceding line.

Direct insertion works for printing characters and `SPC`, but other characters act as editing commands and do not insert themselves. If you need to insert a control character or a character code above 200 octal, you must *quote* it by typing `Control-q` (quoted-insert) first. There are two ways to use `C-q`:

- `Control-q` followed by any non-graphic character (even `C-g`) inserts that character.
- `Control-q` followed by three octal digits inserts the character with the specified character code.

A numeric argument to `C-q` specifies how many copies of the quoted character should be inserted (see chapter 5 [Arguments], page 23).

4.2 Continuation Lines

If you add too many characters to one line, without breaking it with a RET, the line will grow to occupy two (or more) lines on the screen, with a '\' at the extreme right margin of all but the last of them. The '\' says that the following screen line is not really a distinct line in the text, but just the *continuation* of a line too long to fit the screen. Sometimes it is nice to have Emacs insert newlines automatically when a line gets too long; for this, use Auto Fill mode (see section 22.6 [Filling], page 106).

Continuation can be turned off for a particular buffer by setting the variable **truncate-lines** to non-**nil** in that buffer. Then, lines are *truncated*: the text that goes past the right margin does not appear at all. '$' is used in the last column instead of '\' when truncation is in effect. Truncation instead of continuation also happens whenever horizontal scrolling is in use, and optionally whenever side-by-side windows are in use (see chapter 19 [Windows], page 91). **truncate-lines** is automatically local in all buffers. When a buffer is created, its value of **trucate-lines** is initialized from the value of **default-truncate-lines**, normally **nil**.

4.3 Changing the Location of Point

To do more than insert characters, you have to know how to move point (see section 1.1 [Point], page 5). Here are a few of the commands for doing that.

`C-a`	Move to the beginning of the line (**beginning-of-line**).
`C-e`	Move to the end of the line (**end-of-line**).
`C-f`	Move forward one character (**forward-char**).
`C-b`	Move backward one character (**backward-char**).
`C-n`	Move down one line, vertically (**next-line**). If you start in the middle of one line, you end in the middle of the next. From the last line of text, `C-n` creates a new line and moves onto it.
`C-p`	Move up one line, vertically (**previous-line**).
`C-l`	Clear the screen and reprint everything (**recenter**).
`C-t`	Transpose two characters, the ones before and after the cursor (**transpose-chars**).
`M-<`	Move to the top of the buffer (**beginning-of-buffer**). With numeric argument n, move to $n/10$ of the way from the top. See chapter 5 [Arguments], page 23, for more information on numeric arguments.
`M->`	Move to the end of the buffer (**end-of-buffer**).

`M-x goto-char`
: Read a number *n* and move cursor to character number *n*. Position 1 is the beginning of the buffer.

`M-x goto-line`
: Read a number *n* and move cursor to line number *n*. Line 1 is the beginning of the buffer.

`C-x C-n`
: Set current column as goal column for `C-n` and `C-p`. Henceforth, those commands move to this fixed column in the line moved to (**set-goal-column**).

`C-u C-x C-n`
: Cancel the goal column. Henceforth, `C-n` and `C-p` try to stay in the same column, as usual.

If you set the variable **track-eol** to a non-**nil** value, then `C-n` and `C-p` when at the end of the starting line move to the end of the line. Normally, **track-eol** is **nil**.

4.4 Erasing Text

`DEL`
: Delete the character before the cursor (**delete-backward-char**).

`C-d`
: Delete the character after the cursor (**delete-forward-char**).

`C-k`
: Kill to the end of the line (**kill-line**).

You already know about the `DEL` key which deletes the character before the cursor. Another key, `Control-d`, deletes the character after the cursor, causing the rest of the text on the line to shift left. If `Control-d` is typed at the end of a line, that line and the next line are joined together.

To erase a larger amount of text, use the `Control-k` key, which kills a line at a time. If `Control-k` is done at the beginning or middle of a line, it kills all the text up to the end of the line. If `Control-k` is done at the end of a line, it joins that line and the next line.

See section 10.1 [Killing], page 41, for more flexible ways of killing text.

4.5 Files

The commands above are sufficient for creating and altering text in an Emacs buffer; the more advanced Emacs commands just make things easier. But to keep any text permanently you must put it in a *file*. Files are named units of text which are stored by the operating system for you to retrieve later by name. To look at or use the contents of a file in any way, including editing the file with Emacs, you must specify the file name.

Consider a file named /usr/rms/foo.c. To edit this file in Emacs, type

```
    C-x C-f /usr/rms/foo.c RET
```

Here the file name is given as an *argument* to the command `C-x C-f` (find-file). `RET` is used to terminate the argument. Emacs obeys the command by *visiting* the file: creating a buffer, copying the contents of the file into the buffer, and then displaying the buffer for you to edit. You can make changes in it, and then *save* the file by typing `C-x C-s` (save-buffer). This makes the changes permanent by copying the altered contents of the buffer back into the file /usr/rms/foo.c. Until then, the changes are only inside your Emacs, and the file foo.c is not changed.

To create a file, just visit the file with `C-x C-f` as if it already existed. Emacs will make an empty buffer in which you can insert the text you want to put in the file. When you save your text with `C-x C-s`, the file will be created.

Of course, there is a lot more to learn about using files. See chapter 16 [Files], page 71.

4.6 Help

If you forget what a key does, you can find out with the Help character, which is `C-h`. Type `C-h k` followed by the key you want to know about; for example, `C-h k C-n` tells you all about what `C-n` does. `C-h` is a prefix key; `C-h k` is just one of its subcommands. The other subcommands of `C-h` provide different kinds of help. See chapter 8 [Help], page 33.

4.7 Blank Lines

`C-o`	Insert one or more blank lines after the cursor.
`C-x C-o`	Delete all but one of many consecutive blank lines.

When you want to insert a new line of text before an existing line, you can do it by typing the new line of text, followed by `RET`. However, it may be easier to see what you are doing if you first make a blank line and then insert the desired text into it. This is easy to do using the key `C-o` (Customizers: this is bound to the command **open-line**), which inserts a newline after point but leaves point in front of the newline. After `C-o`, type the text for the new line. `C-o F O O` has the same effect as `F O O RET`, except for the final location of point.

You can make several blank lines by typing `C-o` several times, or by giving it an argument to tell it how many blank lines to make. See chapter 5 [Arguments], page 23, for how.

If you have many blank lines in a row and want to get rid of them, use `C-x C-o` (the command **delete-blank-lines**). When point is on a blank line which is adjacent to at least one other blank line, `C-x C-o` deletes all but one of the consecutive blank lines, leaving exactly one. With point on a blank line with no other blank line adjacent to it, the sole blank line is deleted, leaving none. When point is on a nonblank line, `C-x C-o` deletes any blank lines following that nonblank line.

4.8 Cursor Position Information

If you are accustomed to other display editors, you may be surprised that Emacs does not always display the page number or line number of point in the mode line. This is because the text is stored in a way that makes it difficult to compute this information. Displaying them all the time would be intolerably slow. They are not needed very often in Emacs anyway, but there are commands to print them.

`C-x =`
: Print character code of character after point, character position of point, and column of point (**what-cursor-position**).

`M-x what-page`
: Print page number of point, and line number within page.

`M-x what-line`
: Print line number of point in the buffer.

`M-=`
: Print number of lines in the current region.

The command `C-x =` (**what-cursor-position**) can be used to find out the column that the cursor is in, and other miscellaneous information about point. It prints a line in the echo area that looks like this:

```
Char: x (170)  dot=24182 of 469463(5%)  x=40
```

(In fact, this is the output produced when point is before the 'x=40' in the example.)

The two values after '`Char:`' are describe the character following point, first by showing it and second by giving its octal character code.

'`dot=`' is followed by the position of point expressed as a character count. The front of the buffer counts as position 1, one character later as 2, and so on. The next, larger number is the total number of characters in the buffer. Afterward in parentheses comes the position expressed as a percentage of the total size.

'`x=`' is followed by the horizontal position of point, in columns from the left edge of the window.

If the buffer has been narrowed, making some of the text at the beginning and the end temporarily invisible, `C-x =` prints additional text describing the current visible range. For example, it might say

```
Char: x (170)  dot=24182 of 469463(5%) <23868 - 25074>  x=40
```

where the two extra numbers give the smallest and largest character position that point is allowed to assume. The characters between those two positions are the visible ones. See chapter 28 [Narrowing], page 153.

If point is at the end of the buffer (or the end of the visible part), `C-x =` omits any description of the character after point. The output looks like

```
dot=469463 of 469463(5%)  x=0
```

Usually '`x=0`' at the end, because the text usually ends with a newline.

There are two commands for printing line numbers. `M-x what-line` counts lines from the beginning of the file and prints the line number point is on. The first line of the file is line number 1. By contrast, `M-x what-page` counts pages from the beginning of the file, and counts lines within the page, printing both of them. See section 22.5 [Pages], page 105.

While on this subject, we might as well mention `M-=` (count-lines-region), which prints the number of lines in the region (see chapter 9 [Mark], page 37), since there is no other obvious place to stick it. See section 22.5 [Pages], page 105, for the command `C-x l` which counts the lines in the current page.

5. Numeric Arguments

Any Emacs command can be given a *numeric argument*. Some commands interpret the argument as a repetition count. For example, giving an argument of ten to the key `C-f` (the command **forward-char**, move forward one character) moves forward ten characters. With these commands, no argument is equivalent to an argument of one. Negative arguments are allowed. Often they tell a command to move or act backwards.

Some commands care only about whether there is an argument, and not about its value. For example, the command `M-q` (**fill-paragraph**) with no argument fills text; with an argument, it justifies the text as well. (See section 22.6 [Filling], page 106, for more information on `M-q`.)

Some commands use the value of the argument as a repeat count, but do something peculiar when there is no argument. For example, the command `C-k` (**kill-line**) with argument *n* kills *n* lines, including their terminating newlines. But `C-k` with no argument is special: it kills the text up to the next newline, or, if point is right at the end of the line, it kills the newline itself. Thus, two `C-k` commands with no arguments can kill a nonblank line, just like `C-k` with an argument of one. (See section 10.1 [Killing], page 41, for more information on `C-k`.)

If your terminal keyboard has a `META` key, the easiest way to specify a numeric argument is to type digits and/or a minus sign while holding down the the `META` key. For example,

```
M-5 C-n
```

would move down five lines. The characters `Meta-1`, `Meta-2`, etc., and `Meta--`, do this because they are keys bound to commands (**digit-argument** and **negative-argument**) that are defined to contribute to an argument for the next command.

Another way of specifying an argument is to use the `C-u` (**universal-argument**) command followed by the digits of the argument. With `C-u`, you can type the argument digits without holding down shift keys. To type a negative argument, start with a minus sign. Just a minus sign normally means -1. `C-u` works on all terminals.

`C-u` followed by a character which is neither a digit nor a minus sign has the special meaning of "multiply by four". It multiplies the argument for the next command by four. `C-u` twice multiplies it by sixteen. Thus, `C-u C-u C-f` moves forward sixteen characters. This is a good way to move forward "fast", since it moves about 1/5 of a line in the usual size window and font. Other useful combinations are `C-u C-n`, `C-u C-u C-n` (move down a good fraction of a screen), `C-u C-u C-o` (make "a lot" of blank lines), and `C-u C-k` (kill four lines). With commands like `M-q` that care whether there is an argument but not what the value is, `C-u` is a good way of saying, "Let there be an argument."

A few commands treat a plain `C-u` differently from an ordinary argument. A few others may treat an argument of just a minus sign differently from an argument of `-1`. These unusual cases will be described when they come up; they are always for reasons of convenience of use of the individual command.

6. The Minibuffer

The *minibuffer* is the facility used by Emacs commands to read arguments more complicated than a single number. Minibuffer arguments can be file names, buffer names, Lisp function names, Emacs command names, Lisp expressions, and many other things, depending on the command reading the argument.

When the minibuffer is in use, it appears in the echo area, and the terminal's cursor moves there. The beginning of the minibuffer line displays a *prompt* which says what kind of input you should supply and how it will be used. Often this prompt is derived from the name of the command that the argument is for. The prompt normally ends with a colon.

Sometimes a *default argument* appears in parentheses after the colon; it too is part of the prompt. The default will be used as the argument value if you enter an empty argument (e.g., just type `RET`). For example, commands that read buffer names always show a default, which is the name of the buffer that will be used if you type just `RET`.

The simplest way to give a minibuffer argument is to type the text you want, terminated by `RET` which exits the minibuffer. You can get out of the minibuffer, canceling the command that it was for, by typing `C-g`.

Since the minibuffer uses the screen space of the echo area, it can conflict with other ways Emacs customarily uses the echo area. Here is how Emacs handles such conflicts:

- If a command gets an error while you are in the minibuffer, this does not cancel the minibuffer. However, the echo area is needed for the error message and therefore the minibuffer itself is hidden for a while. It comes back after a few seconds.
- If in the minibuffer you use a command whose purpose is to print a message in the echo area, such as `C-x =`, the message is printed normally, and the minibuffer is hidden until the next time you type a character.
- Echoing of commands does not take place while the minibuffer is in use.

6.1 Minibuffers for File Names

Sometimes the minibuffer starts out with text in it. For example, when you are supposed to give a file name, the minibuffer starts out containing the *default directory*, which ends with a slash.

This is to inform you which directory the file will be found in if you do not specify a directory. For example, the minibuffer might start out with

```
    Find File: /u2/emacs/src/
```

where 'Find File: ' is the prompt. Typing buffer.c specifies the file /u2/emacs/src/buffer.c. To find files in nearby directories, use ..; thus, if you type ../lisp/simple.el, you will find the file /u2/emacs/lisp/simple.el. Alternatively, you can kill with M-DEL the directory names you don't want.

You can also type an absolute file name, one starting with a slash or a tilde, ignoring the default directory. For example, to find the file /etc/termcap, just type the name, giving

```
    Find File: /u2/emacs/src//etc/termcap
```

Two slashes in a row are not normally meaningful in Unix file names, but they are allowed in GNU Emacs. They mean, "ignore everything before the second slash in the pair." Thus, '/u2/emacs/src/' is ignored, and you get the file /etc/termcap.

If you set insert-default-directory to nil, the default directory is not inserted in the minibuffer.

6.2 Editing in the Minibuffer

The minibuffer is an Emacs buffer (albeit a peculiar one), and the usual Emacs commands are available for editing the text of an argument you are entering.

Since RET in the minibuffer is defined to exit the minibuffer, inserting a newline into the minibuffer must be done with C-o or with C-q LFD. (Recall that a newline is really the LFD character.)

The minibuffer has its own window which always has space on the screen but acts as if it were not there when the minibuffer is not in use. When the minibuffer is in use, its window is just like the others; you can switch to another window with C-x o, edit text in other windows and perhaps even visit more files, before returning to the minibuffer to submit the argument. You can kill text in another window, return to the minibuffer window, and then yank the text to use it in the argument. See chapter 19 [Windows], page 91.

There are some restrictions on the use of the minibuffer window, however. You cannot switch buffers in it—the minibuffer and its window are permanently attached. Also, you cannot split the minibuffer window.

Recursive use of the minibuffer is supported by Emacs. However, it is easy to do this by accident (because of autorepeating keyboards, for example) and get confused. Therefore, most Emacs commands that use the minibuffer refuse to operate if the minibuffer window is selected. If the minibuffer is active but you have switched to a different window, recursive use of the minibuffer is allowed—if you know enough to try to do this, you probably will not get confused.

If you set the variable **enable-recursive-minibuffers** to be non-**nil**, recursive use of the minibuffer is always allowed.

6.3 Completion

Often, the minibuffer provides a *completion* facility. This means that you type enough of the argument to determine the rest, based on Emacs's knowledge of which arguments make sense, and Emacs visibly fills in the rest, or as much as can be determined from the part you have typed.

When completion is available, certain keys—`TAB`, `RET`, and `SPC`—are redefined to complete an abbreviation present in the minibuffer into a longer string that it stands for, by matching it against a set of *completion alternatives* provided by the command reading the argument.

For example, when the minibuffer is being used by `Meta-x` to read the name of a command, it is given a list of all available Emacs command names to complete against. The completion keys match the text in the minibuffer against all the command names, find any additional characters of the name that are implied by the ones already present in the minibuffer, and add those characters to the ones you have given.

Here is a list of all the completion commands, defined in the minibuffer when completion is available.

`TAB`	Complete the text in the minibuffer as much as possible (**minibuffer-complete**).
`SPC`	Complete the text in the minibuffer but don't add or fill out more than one word (**minibuffer-complete-word**).
`RET`	Submit the text in the minibuffer as the argument, possibly completing first as described below (**minibuffer-complete-and-exit**).

?	Print a list of all possible completions of the text in the minibuffer (**minibuffer-list-completions**).

Completion is very hard to explain but easy to understand once you have seen it in operation. If you type `Meta-x au TAB`, the `TAB` looks for alternatives (in this case, command names) that start with 'au'. In this case, there are only two: **auto-fill-mode** and **auto-save-mode**. These are the same as far as **auto-**, so the 'au' in the minibuffer changes to '`auto-`'.

If you go on to type `f TAB`, this second `TAB` sees '`auto-f`'. The only command name starting this way is **auto-fill-mode**, so that is the completion. You have now have '`auto-fill-mode`' in the minibuffer after typing just au `TAB f TAB`. Note that `TAB` has this effect because in the minibuffer it is bound to the function **minibuffer-complete** when completion is supposed to be done.

`SPC` completes much like `TAB`, but never adds goes beyond the next hyphen. If you have '`auto-f`' in the minibuffer and type `SPC`, it finds that the completion is '`auto-fill-mode`', but it stops completing after '`fill-`'. This gives '`auto-fill-`'. Another `SPC` at this point completes all the way to '`auto-fill-mode`'. `SPC` in the minibuffer runs the function **minibuffer-complete-word** when completion is available.

There are three different ways that `RET` can work in completing minibuffers, depending on how the argument will be used.

- *Strict* completion is used when it is meaningless to give any argument except one of the known alternatives. For example, when `C-x k` reads the name of a buffer to kill, it is meaningless to give anything but the name of an existing buffer. In strict completion, `RET` refuses to exit if the text in the minibuffer does not complete to an exact match.
- *Cautious* completion is similar to strict completion, except that `RET` exits only if the text was an exact match already, not needing completion. If the text is not an exact match, `RET` does not exit, but it does complete the text. If it completes to an exact match, a second `RET` will exit.

  Cautious completion is used for reading file names for files that must already exist.
- *Permissive* completion is used when any string whatever is meaningful, and the list of completion alternatives is just a guide. For example, when `C-x C-f` reads the name of a file to visit, any file name is allowed, in case you want to create a file. In permissive completion, `RET` takes the text in the minibuffer exactly as given, without completing it.

When completion is done on file names, certain file names are usually ignored. The variable **completion-ignored-extensions** contains a list of strings; a file whose name ends in any of those strings is ignored as a possible completion. The standard value of this variable is (".o" ".elc" "~"), which is designed to allow '`foo`' to complete to '`foo.c`' even though '`foo.o`' exists as well. If the only possible completions are files that end in "ignored" strings, then they are not ignored.

6.4 Repeating Minibuffer Commands

Every command that uses the minibuffer at least once is recorded on a special history list, together with the values of the minibuffer arguments, so that you can repeat the command easily. In particular, every use of `Meta-x` is recorded, since `M-x` uses the minibuffer to read the command name.

`C-x ESC` Re-execute a recent minibuffer command.

`C-x ESC` (**repeat-complex-command**) is used to re-execute a recent minibuffer-using command. With no argument, it repeats the last such command. A numeric argument specifies which command to repeat; one means the last one, and larger numbers specify earlier ones.

`C-x ESC` works by turning the previous command into a Lisp expression and then entering a minibuffer initialized with the text for that expression. If you type just `RET`, the command is repeated as before. You can also change the command by editing the Lisp expression. Whatever expression you finally submit is what will be executed. The repeated command does not go on the command history itself; `C-x ESC` does not alter the command history.

The list of previous minibuffer-using commands is stored as a Lisp list in the variable **command-history**. Each command is stored as a list whose first element is the function called by the command and whose remaining elements are the arguments that were given to it (values, not expressions). If *cmd* is an element of **command-history**, to repeat the command do **(apply (car** *cmd***) (cdr** *cmd***))**.

7. Running Commands by Name

The Emacs commands that are used often or that must be quick to type are bound to keys—short sequences of characters—for convenient use. Other Emacs commands that do not need to be brief are not bound to keys; to run them, you must refer to them by name.

A command name is, by convention, made up of one or words, separated by hyphens; for example, **auto-fill-mode** or **manual-entry**. The use of English words makes the command name easier to remember than a key made up of obscure characters, even though it is more characters to type. Any command can be run by name, even if it is also runnable by keys.

The way to run a command by name is to start with `M-x`, type the command name, and finish it with `RET`. Actually, `M-x` (the command **execute-extended-command**) is using the minibuffer to read the command name.

Emacs uses the minibuffer for reading input for many different purposes; on this occasion, the string '`M-x`' is displayed at the beginning of the minibuffer as a *prompt* to remind you that your input should be the name of a command to be run. See chapter 6 [Minibuffer], page 25, for full information the features of the minibuffer.

You can use completion to enter the command name. For example, the command forward-char can be invoked as an extended command by typing

```
    M-x forward-char RET
```
or
```
    M-x fo TAB c RET
```

Note that forward-char is the same command that you invoke with the key `C-f`. Any command (interactively callable function) defined in Emacs can be called by its name using `M-x` whether or not any keys are bound to it.

If you type `C-g` while the command name is being read, you cancel the `M-x` command and get out of the minibuffer, ending up at top level.

To pass a numeric argument to the command you are invoking with `M-x`, specify the numeric argument before the `M-x`. `M-x` passes the argument along to the function which it calls. The argument value appears in the prompt while the command name is being read.

Normally, when describing a command that is run by name, we omit the RET that is needed to terminate the name. Thus we might speak of `M-x auto-fill-mode` rather than `M-x auto-fill-mode` RET. We mention the RET only when there is a need to emphasize its presence, such as when describing a sequence of input that contains a command name and arguments that follow it.

In this manual, the convention for font usage is that Lisp objects, including command names (which are Lisp symbols), appear in this font, but keyboard input appears in `this font`. This brings up a problem with names of commands that are normally run by name: is the name a piece of Lisp code, or is it a sequence of characters to type? Unfortunately, it is both, but only one of the two fonts can be used. I have chosen to use the Lisp object font when discussing the command, as in auto-fill-mode, but to use the keyboard input font for sequences of input, as in `M-x auto-fill-mode`.

8. Help

Emacs provides extensive help features which revolve around a single character, `C-h`. `C-h` is a prefix key that is used only for documentation-printing commands. The characters that you can type after `C-h` are called *help options*. One help option is `C-h`; that is how you ask for help about using `C-h`.

`C-h C-h` prints a list of the possible help options, and then asks you to go ahead and type the option. It prompts with a string

```
A, C, F, I, K, L, M, N, T, V, W, C-c, C-d or C-h for more help:
```

and you shoul^d type one of those characters. Typing a third `C-h` displays a description of what the options mean; it still waits for you to type an option. To cancel, type `C-g`.

Here is a summary of the defined help commands.

`C-h a` Display list of commands whose names contain a specified string.

`C-h b` Display a table of all key bindings in effect now.

`C-h c` *key*
: Print the name of the command that *key* runs.

`C-h f` *function* `RET`
: Display documentation on the Lisp function named *function*. Note that commands are Lisp functions, so a command name may be used.

`C-h k` *key*
: Display name and documentation of the command *key* runs.

`C-h i` Run Info, the program for browsing documentation files.

`C-h l` Display a description of the last 100 characters you typed.

`C-h m` Display documentation of the current major mode.

`C-h n` Display documentation of Emacs changes, most recent first.

`C-h s` Display current contents of the syntax table, plus an explanation of what they mean.

`C-h t` Display the Emacs tutorial

`C-h v` *var* `RET`
: Display the documentation of the Lisp variable *var*.

`C-h w` *command* `RET`
: Print which keys run the command named *command*.

The most basic C-h options are C-h c and C-h k. C-h c *key* prints in the echo area the name of the command that *key* is bound to. For example, C-h c C-f prints 'forward-char'. Since command names are chosen to describe what the command does, this is a good way to get a very brief description of what *key* does. C-h c runs the command describe-key-briefly.

C-h k *key* is similar but gives more information. It displays the documentation string of the command *key* is bound to as well as its name. This is too big for the echo area, so a window is used for the display.

C-h f (describe-function) reads the name of a Lisp function using the minibuffer, then displays that function's documentation string in a window. Since commands are Lisp functions, you can use this to get the documentation of a command that is known by name. For example,

```
C-h f auto-fill-mode RET
```

displays the documentation of auto-fill-mode. This is the only way to see the documentation of a command that is not bound to any key (one which you would normally call using M-x).

C-h f is also useful for Lisp functions that you are planning to use in a Lisp program. For example, if you have just written the code (make-vector len) and want to be sure that you are using make-vector properly, type C-h f make-vector RET. Because C-h f allows all function names, not just command names, you may find that some of your favorite abbreviations that work in M-x don't work in C-h f. An abbreviation may be unique among command names yet fail to be unique when other function names are allowed.

The function name for C-h f to describe has a default which is used if you type RET leaving the minibuffer empty. The default is the function called by the innermost Lisp expression in the buffer around point, *provided* that is a valid, defined Lisp function name. For example, if point is located following the text '(make-vector (car x)', the innermost list containing point is the one that starts with '(make-vector', so the default is to describe the function make-vector.

C-h f is often useful just to verify that you have the right spelling for the function name. If C-h f mentions a default in the prompt, you have typed the name of a defined Lisp function. If that tells you what you want to know, just type C-g to cancel the C-h f command and go on editing.

C-h v (describe-variable) is like C-h f but describes Lisp variables instead of Lisp functions. Its default is the Lisp symbol around or before point, but only if that is the name of a known Lisp variable. See section 29.2 [Variables], page 159.

A more complicated sort of question to ask is, "What are the commands for working with files?" For this, type `C-h a file RET`, which displays a list of all command names that contain '`file`', such as copy-file, find-file, and so on. With each command name appears a brief description of how to use the command, and what keys you can currently invoke it with. For example, it would say that you can invoke find-file by typing `C-x C-f`. The a in `C-h a` stands for 'Apropos'; `C-h a` runs the Lisp function command-apropos.

Because `C-h a` looks only for functions whose names contain the string which you specify, you must use ingenuity in choosing substrings. If you are looking for commands for killing backwards and `C-h a kill-backwards RET` doesn't reveal any, don't give up. Try just `kill`, or just `backwards`, or just `back`. Be persistent. Pretend you are playing Adventure.

Here is a set of arguments to give to apropos that covers many classes of Emacs commands, since there are strong conventions for naming the standard Emacs commands. By giving you a feel for the naming conventions, this set should also serve to aid you in developing a technique for picking apropos strings.

> char, line, word, sentence, paragraph, region, page, sexp, list, defun, buffer, screen, window, file, dir, register, mode, beginning, end, forward, backward, next, previous, up, down, search, goto, kill, delete, mark, insert, yank, fill, indent, case, change, set, what, list, find, view, describe.

When apropos tells you of a command you might want to use, your next question is probably whether any keys are bound to it. `C-h w` (where-is) is the command to tell you this. It reads a command name as a minibuffer argument and prints in the echo area some keys that are bound to that command. Alternatively, it says that the command is not on any keys, which implies that you must use `M-x` to call it.

If something surprising happens, and you are not sure what commands you typed, use `C-h l` (view-lossage). `C-h l` prints the last 100 command characters you typed in. If you see commands that you don't know, you can use `C-h c` to find out what they do.

Emacs has several major modes, each of which redefines a few keys and makes a few other changes in how editing works. `C-h m` (describe-mode) prints documentation on the current major mode, which normally describes all the commands that are changed in this mode.

The other `C-h` options display various files of useful information. `C-h n` (view-emacs-news) displays the file emacs/etc/NEWS, which contains documentation on Emacs changes arranged chronologically. `C-h t` (help-with-tutorial) displays the learn-by-doing Emacs tutorial. `C-h i` (info) runs the Info program, which is used for browsing through structured documentation files. C-h C-c (describe-copying) displays the file emacs/etc/COPYING, which tells you the conditions you should obey in distributing copies of Emacs. C-h C-d (describe-distribution) displays the file emacs/etc/DISTRIB, which tells you how you can order a copy of the latest version of Emacs.

9. The Mark and the Region

There are many Emacs commands which operate on an arbitrary contiguous part of the current buffer. To specify the text for such a command to operate on, you set *the mark* at one end of it, and move point to the other end. The text between point and the mark is called *the region*. You can move point or the mark to adjust the boundaries of the region. It doesn't matter which one is set first chronologically, or which one comes earlier in the text.

Once the mark has been set, it remains until it is set again at another place. The mark remains fixed with respect to the preceding character if text is inserted or deleted in the buffer. Each Emacs buffer has its own mark, so that when you return to a buffer that had been selected previously, it has the same mark it had before.

Many commands that insert text, such as `C-y` (yank) and `M-x Insert Buffer`, position the mark at one end of the inserted text—the opposite end from where point is positioned, so that the region contains the text just inserted.

Here are some commands for setting the mark:

`C-SPC`	Set the mark where point is.
`C-@`	The same.
`C-x C-x`	Interchange mark and point.
`M-@`	Set mark after end of next word. This command and the following three do not move point.
`C-M-@`	Set mark after end of next Lisp expression.
`C-<`	Set mark at beginning of buffer.
`C->`	Set mark at end of buffer.
`M-h`	Put region around current paragraph.
`C-M-h`	Put region around current Lisp defun.
`C-x h`	Put region around entire buffer.
`C-x C-p`	Put region around current page.

For example, if you wish to convert part of the buffer to all upper-case, you can use the `C-x u` (upcase-region) command, which operates on the text in the region. You can first go to the beginning of the text to be capitalized, type `C-SPC` to put the mark there, move to the end, and then type `C-x C-u`. Or, you can set the mark at the end of the text, move to the beginning, and then type `C-x C-u`. Most commands that operate on the text in the region have the word region in their names.

The most common way to set the mark is with the `C-SPC` command (set-mark-command). This sets the mark where point is. Then you can move point away, leaving the mark behind. It is actually incorrect to speak of the character `C-SPC`; there is no such character. When you type `SPC` while holding down control, what you get on most terminals is the character `C-@`. This is the key actually bound to set-mark-command. But unless you are unlucky enough to have a terminal where typing `C-SPC` does not produce `C-@`, you might as well think of this character as `C-SPC`.

Since terminals have only one cursor, there is no way for Emacs to show you where the mark is located. You have to remember. The usual solution to this problem is to set the mark and then use it soon, before you forget where it is. But you can see where the mark is with the command `C-x C-x` (exchange-dot-and-mark) which puts the mark where point was and point where the mark was. The extent of the region is unchanged, but the cursor and point are now at the previous location of the mark.

`C-x C-x` is also useful when you are satisfied with the location of point but want to move the mark; do `C-x C-x` to put point there and then you can move it. A second use of `C-x C-x`, if necessary, puts the mark at the new location with point back at its original location.

9.1 Operating on the Region

Once you have created an active region, you can do many things to the text in it:

- Kill it with `C-w` (see section 10.1 [Killing], page 41).
- Save it in a register with `C-x x` (see chapter 11 [Registers], page 49).
- Save it in a buffer or a file (see section 10.3 [Accumulating Text], page 46).
- Convert case with `C-x C-l` or `C-x C-u` (see section 22.7 [Case], page 108).
- Evaluate it as Lisp code with `M-x eval-region` (see section 24.4 [Lisp Eval], page 134).
- Fill it as text with `M-g` (see section 22.6 [Filling], page 106).
- Print hardcopy with `M-x print-region` (see section 28.2 [Hardcopy], page 155).
- Indent it with `C-x TAB` or `C-M-\` (see chapter 21 [Indentation], page 97).

9.2 Commands to Mark Textual Objects

There are commands for placing the mark on the other side of a certain object such as a word or a list, without having to move there first. `M-@` (mark-word) puts the mark at the end of the next word, while `C-M-@` (mark-sexp) puts it at the end of the next Lisp expression. These characters allow you to save a little typing or redisplay, sometimes.

Other commands set both point and mark, to delimit an object in the buffer. `M-h` (mark-paragraph) moves point to the beginning of the paragraph that surrounds or follows point, and puts the mark at the end of that paragraph (see section 22.4 [Paragraphs], page 104). `M-h` does all that's necessary if you wish to indent, case-convert, or kill a whole paragraph. `C-M-h` (mark-defun) similarly puts point before and the mark after the current or following defun (see section 23.3 [Defuns], page 114). `C-x C-p` (mark-page) puts point before the current page (or the next or previous, according to the argument), and mark at the end (see section 22.5 [Pages], page 105). The mark goes after the terminating page delimiter (to include it), while point goes after the preceding page delimiter (to exclude it). Finally, `C-x h` (mark-whole) sets up the entire buffer as the region, by putting point at the beginning and the mark at the end.

9.3 The Mark Ring

Aside from delimiting the region, the mark is also useful for remembering a spot that you may want to go back to. To make this feature more useful, Emacs remembers 16 previous locations of the mark, in the mark ring. Most commands that set the mark push the old mark onto this ring. To return to a marked location, use `C-u C-@` (or `C-u C-SPC`); this is the command set-mark-command given a numeric argument. This moves point to where the mark was, and restores the mark from the ring of former marks. So repeated use of this command moves point to all of the old marks on the ring, one by one. Enough uses of `C-u C-@` bring point back to where it was originally.

Each buffer has its own mark ring. All editing commands that use the mark ring use the current buffer's mark ring. In particular, `C-u C-SPC` always stays in the same buffer.

Many commands that can move long distances, such as `M-<` (beginning-of-buffer) and `C-M-a` (beginning-of-defun), start by setting the mark and saving the old mark on the mark ring, just as a way of making it possible for you to move to where point was before the command. This is to make it easier for you to move back later. Searches record the starting point except when they do not actually move. You can tell when a command sets the mark because 'Mark Set' is printed in the echo area.

The variable mark-ring-max is the maximum number of entries to keep in the mark ring. If that many entries exist and another one is pushed, the last one in the list is discarded. Repeating `C-u`

C-SPC circulates through the limited number of entries that are currently in the ring.

The variable mark-ring holds the mark ring itself, as a list of marker objects in the order most recent first.

10. Killing and Moving Text

Killing means erasing text and copying it into the *kill ring*, from which it can be retrieved by *yanking* it.

The commonest way of moving or copying text with Emacs is to kill it and later yank it in one or more places. This is very safe because all the text ever killed is remembered, and it is versatile, because the many commands for killing syntactic units can also be used for moving those units. There are also other ways of copying text for special purposes.

Emacs has only one kill ring, so you can kill text in one buffer and yank it in another buffer.

10.1 Deletion and Killing

Most commands which erase text from the buffer save it so that you can get it back if you change your mind, or move or copy it to other parts of the buffer. These commands are known as *kill* commands. The rest of the commands that erase text do not save it; they are known as *delete* commands. The delete commands include `C-d` (delete-char) and `DEL` (delete-backward-char), which delete only one character at a time, and those commands that delete only spaces or newlines. Commands that can destroy significant amounts of nontrivial data generally kill. The commands' names and individual descriptions use the words '`kill`' and '`delete`' to say which they do. If you do a kill or delete command by mistake, you can use the `C-x u` (undo) command to undo it (see chapter 12 [Undo], page 51).

10.1.1 Deletion

`C-d`	Delete next character.
`DEL`	Delete previous character.
`M-\`	Delete spaces and tabs around point.
`M-SPC`	Delete spaces and tabs around point, leaving one space.
`C-x C-o`	Delete blank lines around the current line.
`M-^`	Join two lines by deleting the intervening newline, and any indentation following it.

The most basic delete commands are `C-d` (delete-char) and `DEL` (delete-backward-char). `C-d` deletes the character after point, the one the cursor is "on top of". Point doesn't move. `DEL` deletes the character before the cursor, and moves point back. Newlines can be deleted like any other characters in the buffer; deleting a newline joins two lines. Actually, `C-d` and `DEL` aren't always delete commands; if given an argument, they kill instead, since they can erase more than one character this way.

The other delete commands are those which delete only formatting characters: spaces, tabs and newlines. `M-\` (**delete-horizontal-space**) deletes all the spaces and tab characters before and after point. `M-SPC` (**just-one-space**) does likewise but leaves a single space after point, regardless of the number of spaces that existed previously (even zero).

`C-x C-o` (**delete-blank-lines**) deletes all blank lines after the current line, and if the current line is blank deletes all blank lines preceding the current line as well (leaving one blank line, the current line). `M-^` (**delete-indentation**) joins the current line and the previous line, or the current line and the next line if given an argument, by deleting a newline and all surrounding spaces, possibly leaving a single space. See chapter 21 [Indentation], page 97.

10.1.2 Killing by Lines

`C-k` Kill rest of line or one or more lines.

The simplest kill command is `C-k` (**kill-line**). If given at the beginning of a line, it kills all the text on the line, leaving it blank. If given on a blank line, the blank line disappears. As a consequence, if you go to the front of a non-blank line and type `C-k` twice, the line disappears completely.

More generally, `C-k` kills from point up to the end of the line, unless it is at the end of a line. In that case it kills the newline following the line, thus merging the next line into the current one. Invisible spaces and tabs at the end of the line are ignored when deciding which case applies, so if point appears to be at the end of the line, you can be sure the newline will be killed.

If `C-k` is given a positive argument, it kills that many lines and the newlines that follow them (however, text on the current line before point is spared). With a negative argument, it kills back to a number of line beginnings. An argument of -2 means kill back to the second line beginning. If point is at the beginning of a line, that line beginning doesn't count, so `C-u - 2 C-k` with point at the front of a line kills the two previous lines.

`C-k` with an argument of zero kills all the text before point on the current line.

10.1.3 Other Kill Commands

`C-w`	Kill region (from point to the mark).
`M-d`	Kill word.
`M-DEL`	Kill word backwards.
`C-x DEL`	Kill back to beginning of sentence (see section 22.3 [Sentences], page 103).
`M-k`	Kill to end of sentence.
`C-M-k`	Kill sexp (see section 23.2 [Lists], page 112).
`C-M-DEL`	Kill sexp backwards.
`M-z` *char*	Kill up to next occurrence of *char*.

A kill command which is very general is `C-w` (kill-region), which kills everything between point and the mark. With this command, you can kill any contiguous sequence of characters, if you first set the mark aι one end of them and go to the other end.

A convenient way of killing is combined with searching: `M-z` (zap-to-char) reads a character and kills from point up to (but not including) the next occurrence of that character in the buffer. If there is no next occurrence, killing goes to the end of the buffer. A numeric argument acts as a repeat count. A negative argument means to search backward and kill text before point.

Other syntactic units can be killed: words, with `M-DEL` and `M-d` (see section 22.2 [Words], page 101); sexps, with `C-M-k` (see section 23.2 [Lists], page 112); and sentences, with `C-x DEL` and `M-k` (see section 22.3 [Sentences], page 103).

10.2 Yanking

Yanking is getting back text which was killed. The usual way to move or copy text is to kill it and then yank it one or more times.

`C-y`	Yank last killed text.
`M-y`	Replace re-inserted killed text with the previously killed text.
`M-w`	Save region as last killed text without actually killing it.
`C-M-w`	Append next kill to last batch of killed text.

All killed text is recorded in the *kill ring*, a list of blocks of text that have been killed. There is only one kill ring, used in all buffers, so you can kill text in one buffer and yank it in another buffer.

The command `C-Y` (Yank) reinserts the text of the most recent kill. It leaves the cursor at the end of the text. It sets the mark at the beginning of the text. See chapter 9 [Mark], page 37.

`C-u C-y` leaves the cursor in front of the text, and sets the mark after it. This is only if the argument is specified with just a `C-u`, precisely. Any other sort of argument, including `C-u` and digits, has an effect described below (under "Yanking Earlier Kills").

If you wish to copy a block of text, you might want to use **M-W** (**copy-region-as-kill**), which copies the region into the kill ring without removing it from the buffer. This is approximately equivalent to `C-w` followed by `C-y`, except that `M-w` does not mark the buffer as "modified" and does not temporarily change the screen.

10.2.1 Appending Kills

Normally, each kill command pushes a new block onto the kill ring. However, two or more kill commands in a row combine their text into a single entry, so that a single `C-y` gets it all back as it was before it was killed. This means that you don't have to kill all the text in one command; you can keep killing line after line, or word after word, until you have killed it all, and you can still get it all back at once. (Thus we join television in leading people to kill thoughtlessly.)

Commands that kill forward from point add onto the end of the previous killed text. Commands that kill backward from point add onto the beginning. This way, any sequence of mixed forward and backward kill commands puts all the killed text into one entry without rearrangement. Numeric arguments do not break the sequence of appending kills. For example, suppose the buffer contains

```
This is the first
line of sample text
and here is the third.
```

with point at the beginning of the second line. If you type `C-k C-u 2 M-DEL C-k`, the first `C-k` kills the text '`line of sample text`', `C-u 2 M-DEL` kills '`the first`' with the newline that followed it, and the second `C-k` kills the newline after the second line. The result is that the buffer contains '`This is and here is the third.`' and a single kill entry contains '`the firstRETline of sample textRET`'—all the killed text, in its original order.

If a kill command is separated from the last kill command by other commands (not just numeric arguments), it starts a new entry on the kill ring. But you can force it to append by first typing the command `C-M-w` (**append-next-kill**) in front of it. The `C-M-w` tells the following command, if it is a kill command, to append the text it kills to the last killed text, instead of starting a new entry. With `C-M-w`, you can kill several separated pieces of text and accumulate them to be yanked back in one place.

10.2.2 Yanking Earlier Kills

To recover killed text that is no longer the most recent kill, you need the `Meta-y` (**yank-pop**) command. `M-y` should be used only after a `C-y` or another `M-y`. It takes the text previously yanked and replaces it with the text from an earlier kill. So, to recover the text of the next-to-the-last kill, you first use `C-y` to recover the last kill, and then use `M-y` to replace it with the previous kill.

You can think in terms of a "last yank" pointer which points at an item in the kill ring. Each time you kill, the "last yank" pointer moves to the newly made item at the front of the ring. `C-y` yanks the item which the "last yank" pointer points to. `M-y` moves the "last yank" pointer to a different item, and the text in the buffer changes to match. Enough `M-y` commands can move the pointer to any item in the ring, so you can get any item into the buffer. Eventually the pointer reaches the end of the ring; the next `M-y` moves it to the first item again.

`M-y` can take a numeric argument, which tells it how many items to advance the "last yank" pointer by. A negative argument moves the pointer toward the front of the ring; from the front of the ring, it moves to the last entry and starts moving forward from there.

Once the text you are looking for is brought into the buffer, you can stop doing `M-y` commands and it will stay there. It's just a copy of the kill ring item, so editing it in the buffer does not change what's in the ring. As long as no new killing is done, the "last yank" pointer remains at the same place in the kill ring, so repeating `C-y` will yank another copy of the same old kill.

If you know how many `M-y` commands it would take to find the text you want, you can yank that text in one step using `C-y` with a numeric argument. `C-y` with an argument greater than one restores the text the specified number of entries back in the kill ring. Thus, `C-u 2 C-y` gets the next to the last block of killed text. It is equivalent to `C-y M-y`. `C-y` with a numeric argument starts counting from the "last yank" pointer, and sets the "last yank" pointer to the entry that it yanks.

The length of the kill ring is controlled by the variable **kill-ring-max**; no more than that many blocks of killed text are saved.

10.3 Accumulating Text

Usually we copy or move text by killing it and yanking it, but there are other ways that are useful for copying one block of text in many places, or for copying many scattered blocks of text into one place.

You can accumulate blocks of text from scattered locations either into a buffer or into a file if you like. These commands are described here. You can also use Emacs registers for storing and accumulating text. See chapter 11 [Registers], page 49.

`C-x a` Append region to contents of specified buffer.

`M-x prepend-to-buffer`
: Prepend region to contents of specified buffer.

`M-x copy-to-buffer`
: Copy region into specified buffer.

`M-x insert-buffer`
: Insert contents of specified buffer into current buffer at point.

`M-x append-to-file`
: Append region to contents of specified file, at the end.

To accumulate text into a buffer, use the command `C-x a` *buffername* (append-to-buffer), which inserts a copy of the region into the buffer *buffername*, at the location of point in that buffer. If there is no buffer with that name, one is created. If you append text into a buffer which has been used for editing, the copied text goes into the middle of the text of the buffer, wherever point happens to be in it.

Point in that buffer is left at the end of the copied text, so successive uses of `C-x a` accumulate the text in the specified buffer in the same order as they were copied. Strictly speaking, `C-x a` does not always append to the text already in the buffer; but if `C-x a` is the only command used to alter a buffer, it does always append to the existing text because point is always at the end.

`M-x prepend-to-buffer` is just like `C-x a` except that point in the other buffer is left before the copied text, so successive prependings add text in reverse order. `M-x copy-to-buffer` is similar except that any existing text in the other buffer is deleted, so the buffer is left containing just the text newly copied into it.

You can retrieve the accumulated text from that buffer with `M-x insert-buffer`; this too takes *buffername* as an argument. It inserts a copy of the text in buffer *buffername* into the selected buffer. You could alternatively select the other buffer for editing, perhaps moving text from it by killing or with `C-x a`. See chapter 18 [Buffers], page 85, for background information on buffers.

Instead of accumulating text within Emacs, in a buffer, you can append text directly into a file with `M-x append-to-file`, which takes *file-name* as an argument. It adds the text of the region to the end of the specified file. The file is changed immediately on disk. This commands is normally used with files that are *not* being visited in Emacs. Using them on files that Emacs is visiting can produce confusing results, because the text inside Emacs for those files will not change.

10.4 Rectangles

The rectangle commands affect rectangular areas of the text: all the characters between a certain pair of columns, in a certain range of lines. Commands are provided to kill rectangles, yank killed re-tangles, clear them out, or delete them.

When you must specify a rectangle for a command to work on, you do it by putting the mark at one corner and point at the opposite corner. The rectangle thus specified is called the *region-rectangle* because it is controlled about the same way the region is controlled. But remember that a given combination of point and mark values can be interpreted either as specifying a region or as specifying a rectangle; it is up to the command that uses them to choose the interpretation.

`M-x delete-rectangle`
: Delete the text of the region-rectangle, moving any following text on each line leftward to the left edge of the region-rectangle.

`M-x kill-rectangle`
: Similar, but also save the contents of the region-rectangle as the "last killed rectangle".

`M-x yank-rectangle`
: Yank the last killed rectangle with its upper left corner at point.

`M-x open-rectangle`
: Insert blank space to fill the space of the region-rectangle. The previous contents of the region-rectangle are pushed rightward.

`M-x clear-rectangle`
: Clear the region rectangle by replacing its contents with spaces.

The rectangle operations fall into two classes: commands deleting and moving rectangles, and commands for blank rectangles.

There are two ways to delete a rectangle: you can discard its contents, or save them as the "last killed" rectangle. The commands for these three ways are `M-x delete-rectangle` and `M-x kill-rectangle`. In any case, the portion of each line that falls inside the rectangle's boundaries is deletyed, causing following text (if any) on the line to move left.

Note that "killing" a rectangle is not killing in the usual sense; the rectangle is not stored in the kill ring, but in a special place that can only record the most recent rectangle killed. This is because yanking a rectangle is so different from yanking linear text that different yank commands have to be used and yank-popping is hard to make sense of.

Inserting a rectangle is the opposite of deleting one. All you need to specify is where to put the upper left corner; that is done by putting point there. The rectangle's first line is inserted there, the rectangle's second line is inserted at a point one line vertically down, and so on. The number of lines affected is determined by the height of the saved rectangle.

To insert the last killed rectangle, type `M-x yank-rectangle`.

There are two commands for working with blank rectangles: `M-x clear-rectangle` to blank out existing text, and `M-x open-rectangle` to insert a blank rectangle. Clearing a rectangle is equivalent to deleting it and then inserting as blank rectangle of the same size.

Rectangles can also be copied into and out of registers. See section 11.3 [Rectangle Registers], page 50.

11. Registers

Emacs *registers* are places you can save text or positions for use later. Text saved in a register can be copied into the buffer once or many times; a position saved in a register is used by moving point to that position. Rectangles can also be copied into and out of registers (see section 10.4 [Rectangles], page 47).

Each register has a name, which is a single character. It can store either a piece of text or a position or a rectangle; only one of the three at any given time. Whatever you store in a register remains there until you store something else in that register.

11.1 Saviıg Positions in Registers

`C-x / r`	Save location of point in register *r*.
`C-x j r`	Jump to the location saved in register *r*.

To save the current location of point in a register, choose a name *r* and type `C-x / r`. (`C-x /` runs the command **dot-to-register**.) The register *r* retains the location thus saved until you store something else in that register.

The command `C-x j r` (**register-to-dot**) moves point to the location recorded in register *r*. The register is not affected; it continues to record the same location. You can jump to the same position using the same register any number of times.

11.2 Saving Text in Registers

When you want to insert a copy of the same piece of text frequently, it may be impractical to use the kill ring, since each subsequent kill moves the piece of text farther down on the ring. It becomes hard to keep track of what argument is needed to retrieve the same text with `C-y`. An alternative is to store the text in a register with `C-x x` (**copy-to-register**) and then retrieve it with `C-x g` (**insert-register**).

`C-x x r`	Copy region into register *r*.
`C-x g r`	Insert text contents of register *r*.

`C-x x` *r* stores a copy of the text of the region into the register named *r*. Given a numeric argument, `C-x x` deletes the text from the buffer as well.

`C-x g` *r* inserts in the buffer the text from register *r*. Normally it leaves point before the text and places the mark after, but with a numeric argument it puts point after the text and the mark before.

11.3 Saving Rectangles in Registers

A register can contain a rectangle instead of linear text. The rectangle is represented as a list of strings. See section 10.4 [Rectangles], page 47, for basic information on rectangles and how rectagles in the buffer are specified.

`C-x r` *r*	Copy the region-rectangle into register *r* (**copy-region-to-rectangle**). With numeric argument, delete it as well.
`C-x g` *r*	Insert the rectangle stored in register *r* (if it contains a rectangle).

The `C-x g` command inserts linear text if the register contains that, or inserts a rectangle if the register contains one.

11.4 Getting Information about Registers

`M-x view-register RET` *r*
Display a description of what register *r* contains.

`M-x view-register` reads a register name as an argument and then displays the contents of the specified register.

12. Undoing Changes

Emacs allows all changes made in the text of a buffer to be undone, up to a certain amount of change (8000 characters). Each buffer records changes individually, and the undo command always applies to the current buffer. Usually each editing command makes a separate entry in the undo records, but some commands such as `query-replace` make many entries, and very simple commands such as self-inserting characters are often grouped to make undoing less tedious.

`C-x u`	Undo one batch of changes (usually, one command worth).
`C-_`	The same.

The command `C-x u` or `C-_` (`undo`) is how you undo. The first time you give this command, it undoes the last change. Point moves to the beginning of the text affected by the undo, so you can see what was undone.

Consecutive repetitions of the `C-_` or `C-x u` commands undo earlier and earlier changes, back to the limit of what has been recorded. If all recorded changes have already been undone, the undo command gets an error.

Any command other than an undo command breaks the sequence of undo commands. Starting at this moment, the previous undo commands are considered ordinary changes that can themselves be undone. Thus, you can redo changes you have undone by typing `C-SPC`, `C-f` or any other command that will have no important effect, and then using more undo commands.

If you notice that a buffer has been modified accidentally, the easiest way to recover is to type `C-_` repeatedly until the stars disappear from the front of the mode line. At this time, all the modifications you made have been cancelled. If you do not remember whether you changed the buffer deliberately, type `C-_` once, and when you see the last change you made undone, you will remember why you made it. If it was an accident, leave it undone. If it was deliberate, redo the change as described in the preceding paragraph.

Not all buffers record undo information. Buffers whose names start with spaces don't; these buffers are used internally by Emacs and its extensions to hold text that users don't normally look at or edit. Also, minibuffers, help buffers and documentation buffers don't record undo information.

At most 8000 or so characters of deleted or modified text can be remembered in any one buffer for reinsertion by the undo command. Also, there is a limit on the number of individual insert, delete or change actions that can be remembered.

The reason the undo command has two keys, C-x u and C-_, set up to run it is that it is worthy of a single-character key, but the way to type C-_ on some keyboards is not obvious. C-x u is an alternative that requires no special knowledge of the terminal.

13. Controlling the Display

Since only part of a large buffer fits in the window, Emacs tries to show the part that is likely to be interesting. The display control commands allow you to ask to see a different part of the text. This is also known as *scrolling*.

If a buffer contains text that is too large to fit entirely within a window that is displaying the buffer, Emacs shows a contiguous section of the text. The section shown always contains point. As you change the text, Emacs always tries to keep the same position in the text at the top of the window. A new position moves to the top of the window only if this is necessary to keep point visible, or if you request it explicitly with a display control command.

`C-l`	Clear screen and redisplay, scrolling the selected window to center point vertically within it.
`C-v`	Scroll forwards (a windowful or a few lines).
`M-v`	Scroll backwards.
`C-x <`	Scroll display of lines to the left.
`C-x >`	Scroll display of lines to the right.
`M-r`	Move point to the text at a given vertical position within the window.

The basic display control command is `C-l` (recenter). In its simplest form, with no argument, it clears the entire screen and redisplays all windows, scrolling the selected window so that point is halfway down from the top of the window. Other windows are cleared and redisplayed, but not scrolled.

`C-l` with a numeric argument does not clear the screen; it does nothing except scroll the selected window as specified by the argument. With a positive argument *n*, it repositions text to put point *n* lines down from the top. An argument of zero puts point on the very top line. Point does not move with respect to the text; rather, the text and point move rigidly on the screen. `C-l` with a negative argument puts point that many lines from the bottom of the window. For example, `C-u - 1 C-l` puts point on the bottom line, and `C-u - 5 C-l` puts it five lines from the bottom.

The *scrolling* commands `C-v` and `M-v` let you move the whole display up or down a few lines. `C-v` (scroll-up) with an argument shows you that many more lines at the bottom of the window, moving the text and point up together as `C-l` might. `C-v` with a negative argument shows you more lines at the top of the window. `Meta-v` (scroll-down) is like `C-v`, but moves in the opposite direction.

To read the buffer a windowful at a time, use C-v with no argument. It takes the last line at the bottom of the window and puts it at the top, followed by nearly a whole windowful of lines not visible before. Point is put at the top of the window. Thus, each C-v shows the "next windowful", except for one line of overlap to provide continuity. M-v with no argument moves the same distance backward. The number of lines of overlap across a C-v is controlled by the variable next-screen-context-lines; by default, it is two.

Scrolling happens automatically if point has moved out of the visible portion of the text when it is time to display. Usually the scrolling is done so as to put point vertically centered within the window. However, if the variable scroll-step has a nonzero value, an attempt is made to scroll the buffer by that many lines; if that is enough to bring point back into visibility, that is what is done.

The text in a window can also be scrolled horizontally. This means that each line of text is shifted sideways in the window, and one or more characters at the beginning of each line are not displayed at all. When a window has been scrolled horizontally in this way, text lines are truncated rather than continued (see section 4.2 [Continuation Lines], page 18), with a '$' appearing in the first column when there is text truncated to the left, and in the last column when there is text truncated to the right.

The command C-x < (scroll-left) scrolls the selected window to the left by one column, or *n* columns with argument *n*. C-x > (scroll-right) scrolls in right one or more columns. The window cannot be scrolled any farther to the right once it is displaying normally (with each line starting at the window's right margin); attempting to do so has no effect.

The commands described above all change the position of point on the screen, carrying the text with it. Another command moves point the same way but leaves the text fixed. It is Meta-r (move-to-window-line). With no argument, it puts point at the beginning of the line at the center of the window. An argument is used to specify the line to put point on, counting from the top if the argument is positive, or from the bottom if it is negative. Thus, M-0 M-r moves point to the text at the top of the window. Meta-r never causes any text to move on the screen; it causes point to move with respect to the screen and the text.

14. Searching and Replacement

Like other editors, Emacs has commands for searching for an occurrence of a string. The principal search command is unusual in that it is *incremental*; it begins to search before you have finished typing the search string. There are also nonincremental search commands more like those of other editors.

Besides the usual **replace-string** command that finds all occurrences of one string and replaces them with another, Emacs has a fancy replacement command called **query-replace** which asks interactively which occurrences to replace.

14.1 Incremental Search

An incremental search begins searching as soon as you type the first character of the search string. As you type in the search string, Emacs shows you where the string (as you have typed it so far) would be found. When you have typed enough characters to identify the place you want, you can stop. Depending on what you will do next, you may or may not need to terminate the search explicitly with an ESC first.

`C-s`	Search forward.
`C-r`	Search backward.

The command to search is `C-s` (**isearch-forward**). `C-s` reads characters from the keyboard and positions the cursor at the first occurrence of the characters that you have typed. If you type `C-s` and then `F`, the cursor moves right after the first '`F`'. Type an `O`, and see the cursor move to after the first '`FO`'. After another `O`, the cursor is after the first '`FOO`' after the place where you started the search. Meanwhile, the search string '`FOO`' has been echoed in the echo area.

If you make a mistake in typing the search string, you can erase characters with `DEL`. Each `DEL` cancels the last character of search string. This does not happen until Emacs is ready to read another input character; first it must either find, or fail to find, the character you want to erase. If you do not want to wait for this to happen, use `C-g` as described below.

When you are satisfied with the place you have reached, you can type `ESC`, which stops searching, leaving the cursor where the search brought it. Also, any command not specially meaningful in searches stops the searching and is then executed. Thus, typing `C-a` would exit the search and then move to the beginning of the line. `ESC` is necessary only if the next command you want to type is a printing character, DEL, ESC, or another control character that is special within searches (`C-q`, `C-w`, `C-r`, `C-s` or `C-k`).

Sometimes you search for '`FOO`' and find it, but not the one you expected to find. There was a second '`FOO`' that you forgot about, before the one you were looking for. In this event, type another `C-s` to move to the next occurrence of the search string. This can be done any number of times. If you overshoot, you can cancel some `C-s` characters with DEL.

After you exit a search, you can search for the same string again by typing just `C-s C-s`: the first `C-s` is the key that invokes incremental search, and the second `C-s` means "search again".

If your string is not found at all, the echo area says '`Failing I-Search`'. The cursor is after the place where Emacs found as much of your string as it could. Thus, if you search for '`FOOT`', and there is no '`FOOT`', you might see the cursor after the '`FOO`' in '`FOOL`'. At this point there are several things you can do. If your string was mistyped, you can rub some of it out and correct it. If you like the place you have found, you can type ESC or some other Emacs command to "accept what the search offered". Or you can type `C-g`, which removes from the search string the characters that could not be found (the '`T`' in '`FOOT`'), leaving those that were found (the '`FOO`' in '`FOOT`'). A second `C-g` at that point cancels the search entirely, returning point to where it was when the search started.

The `C-g` "quit" character does special things during searches; just what it does depends on the status of the search. If the search has found what you specified and is waiting for input, `C-g` cancels the entire search. The cursor moves back to where you started the search. If `C-g` is typed when there are characters in the search string that have not been found—because Emacs is still searching for them, or because it has failed to find them—then the search string characters which have not been found are discarded from the search string. With them gone, the search is now successful and waiting for more input, so a second `C-g` will cancel the entire search.

To search for a control character such as `C-s` or DEL or ESC, you must quote it by typing `C-Q` first. This function of `C-Q` is analogous to its meaning as an Emacs command: it causes the following character to be treated the way a graphic character would normally be treated in the same context.

You can change to searching backwards with `C-r`. If a search fails because the place you started was too late in the file, you should do this. Repeated `C-r` keeps looking for more occurrences backwards. A `C-s` starts going forwards again. `C-r` can be rubbed out just like anything else. If you know that you want to search backwards, you can use `C-r` instead of `C-s` to start the search, because `C-r` is also a key running a command (**isearch-reverse**) to search backward.

The characters `C-y` and `C-w` can be used in incremental search to grab text from the buffer into the search string. This makes it convenient to search for another occurrence of text at point. `C-w` grabs the word after point and makes it part of the search string; `C-y` grabs the rest of the line onto the end of the search string. Both commands advance over the text that is grabbed. To search for the next occurrence of the text that was grabbed, type `C-s`.

All the characters special in incremental search can be changed by setting the following variables:

search-delete-char
: Character to delete from incremental search string (normally `DEL`).

search-exit-char
: Character to exit incremental search (normally `ESC`).

search-quote-char
: Character to quote special characters for incremental search (normally `C-q`).

search-repeat-char
: Character to repeat incremental search forwards (normally `C-s`).

search-reverse-char
: Character to repeat incremental search backwards (normally `C-r`).

search-yank-line-char
: Character to pull rest of line from buffer into search string (normally `C-y`).

search-yank-word-char
: Character to pull next word from buffer into search string (normally `C-w`).

14.1.1 Slow Terminal Incremental Search

Incremental search on a slow terminal uses a modified style of display that is designed to take less time. Instead of redisplaying the buffer at each place the search gets to, it creates a new single-line window and uses that to display the line that the search has found. The single-line window comes into play as soon as point gets outside of the text that is already on the screen.

When the search is terminated, the single-line window is removed. Only at this time is the window in which the search was done redisplayed to show its new value of point.

The slow terminal style of display is used when the terminal baud rate is less than or equal to the value of the variable **isearch-slow-speed**, initially 1200.

14.2 Nonincremental Search

Emacs also has conventional nonincremental search commands, which require you to type the entire search string before searching begins.

`C-s ESC` *string* `RET`
: Search for *string*.

`C-r ESC` *string* `RET`
: Search backward for *string*.

To do a nonincremental search, first type `C-s ESC`. This enters the minibuffer to read the search string; terminate the string with `RET`, and then the search is done. If the string is not found the search command gets an error.

The way `C-s ESC` works is that the `C-s` invokes incremental search, which is specially programmed to invoke nonincremental search if the argument you give it is empty. (Such an empty argument would otherwise be useless.) `C-r ESC` also works this way.

Forward and backward nonincremental searches are implemented by the commands search-forward and search-backward. These commands may be bound to keys in the usual manner. The reason that they are reached by special-case code in incremental search is because `C-s ESC` is the traditional sequence of characters used in Emacs to invoke nonincremental search.

However, nonincremental searches performed using `C-s ESC` do not call search-forward right away. The first thing done is to see if the next character is `C-w`, which requests a word search.

14.3 Word Search

Word search searches for a sequence of words without regard to how the words are separated. More precisely, you type a string of many words, using single spaces to separate them, and the string can be found even if there are multiple spaces, newlines or other punctuation between the words.

Word search is useful in editing documents formatted by text formatters. If you edit while looking at the printed, formatted version, you can't tell where the line breaks are in the source file. With word search, you can search without having to know them.

`C-s ESC C-w` *words* `RET`
: Search for *words*, ignoring differences in punctuation.

`C-r ESC C-w` *words* `RET`
: Search backward for *words*, ignoring differences in punctuation.

Word search is a special case of nonincremental search and is invoked with `C-s ESC C-w`. This is followed by the search string, which must always be terminated with `RET`. Being nonincremental, this search does not start until the argument is terminated. It works by constructing a regular expression and searching for that. See section 14.4 [Regexp Search], page 59.

A backward word search can be done by `C-r ESC C-w`.

Forward and backward word searches are implemented by the commands word-search-forward and word-search-backward. These commands may be bound to keys in the usual manner. The reason that tney are reached by special-case code in incremental and nonincremental search is because `C-s ESC C-w` is the traditional Emacs sequence of keys to use to do a word search.

14.4 Regular Expression Search

A *regular expression* (*regexp*, for short) is a pattern that denotes a set of strings, possibly an infinite set. Searching for matches for a regexp is a very powerful operation that editors on Unix systems have traditionally offered. In GNU Emacs, you can search for the next match for a regexp either incrementally or not.

Incremental search for a regexp is done by typing `C-M-s` (isearch-forward-regexp). This command reads a search string incrementally just like `C-s`, but it treats the search string as a regexp rather than looking for an exact match against the text in the buffer. Each time you add text to the search string, you make the regexp longer, and the new regexp is searched for.

Nonincremental search for a regexp is done by the functions re-search-forward and re-search-backward. You can invoke these with `M-x`, or bind them to keys. Also, you can call re-search-forward by way of incremental regexp search with `C-M-s ESC`.

14.5 Syntax of Regular Expressions

Regular expressions have a syntax in which a few characters are special constructs and the rest are *ordinary*. An ordinary character is a simple regular expression which matches that character and nothing else. The special characters are '`$`', '`^`', '`.`', '`*`', '`[`', '`]`' and '`\`'. Any other character appearing in a regular expression is ordinary, unless a '`\`' precedes it.

For example, '`f`' is not a special character, so it is ordinary, and therefore '`f`' is a regular expression that matches the string '`f`' and no other string. (It does *not* match the string '`ff`'.) Likewise, '`o`' is a regular expression that matches only '`o`'.

Any two regular expressions *a* and *b* can be concatenated. The result is a regular expression which matches a string if *a* matches some amount of the beginning of that string and *b* matches the rest of the string.

As a simple example, we can concatenate the regular expressions '`f`' and '`o`' to get the regular expression '`fo`', which matches only the string '`fo`'. Still trivial.

Note: for historical compatibility, special characters are treated as ordinary ones if they are in contexts where their special meanings make no sense. For example, '`*foo`' treats '`*`' as ordinary since there is no preceding expression on which the '`*`' can act. It is poor practice to depend on this behaviour; better to quote the special character anyway, regardless of where is appears.

`.` is a special character that matches anything except a newline. Using concatenation, we can make regular expressions like '`a.b`' which matches any three-character string which begins with '`a`' and ends with '`b`'.

`*` is not a construct by itself; it is a suffix, which means the preceding regular expression is to be repeated as many times as possible. In '`fo*`', the '`*`' applies to the '`o`', so '`fo*`' matches '`f`' followed by any number of '`o`'s. The case of zero '`o`'s is allowed: '`fo*`' does match '`f`'.

'`*`' always applies to the *smallest* possible preceding expression. Thus, '`fo*`' has a repeating '`o`', not a repeating '`fo`'.

The matcher processes a '`*`' construct by matching, immediately, as many repetitions as can be found. Then it continues with the rest of the pattern. If that fails, backtracking occurs, discarding some of the matches of the '`*`'-modified construct in case that makes it possible to match the rest of the pattern. For example, matching '`c[ad]*ar`' against the string '`caddaar`', the '`[ad]*`' first matches '`addaa`', but this does not allow the next '`a`' in the pattern to match. So the last of the matches of '`[ad]`' is undone and the following '`a`' is tried again. Now it succeeds.

`[ ... ]` '`[`' begins a *character set*, which is terminated by a '`]`'. In the simplest case, the characters between the two form the set. Thus, '`[ad]`' matches either '`a`' or '`d`', and '`[ad]*`' matches any string of '`a`' and '`d`' (including the empty string), from which it follows that '`c[ad]*r`' matches '`car`', etc.

Character ranges can also be included in a character set, by writing two characters with a '`-`' between them. Thus, '`[a-z]`' matches any lower-case letter. Ranges may be intermixed freely with individual characters, as in '`[a-z$%.]`', which matches any lower case letter or '`$`', '`%`' or period.

Note that the usual special characters are not special any more inside a character set. A completely different set of special characters exists inside character sets: '`]`', '`-`' and '`^`'.

To include a '`]`' in a character set, you must make it the first character. For example, '`[]a]`' matches '`]`' or '`a`'. To include a '`-`', you must use it in a context where it cannot possibly indicate a range: that is, as the first character, or immediately after a range.

`[^ ... ]`

'`[^`' begins a *complement character set*, which matches any character except the ones specified. Thus, '`[^a-z0-9A-Z]`' matches all characters *except* letters and digits.

'`^`' is not special in a character set unless it is the first character. The character following the '`^`' is treated as if it were first (it may be a '`-`' or a '`]`').

`^` is a special character that matches the empty string, but only if at the beginning of a line in the text being matched. Otherwise it fails to match anything. Thus, '`^foo`' matches a '`foo`' which occurs at the beginning of a line.

`$` is similar to '`^`' but matches only at the end of a line. Thus, '`xx*$`' matches a string of one '`x`' or more at the end of a line.

`\` has two functions: it quotes the above special characters (including '`\`'), and it introduces additional special constructs.

Because '`\`' quotes special characters, '`\$`' is a regular expression which matches only '`$`', and '`\[`' is a regular expression which matches only '`[`', and so on.

For the most part, '`\`' followed by any character matches only that character. However, there are several exceptions: characters which, when preceded by '`\`', are special constructs. Such characters are always ordinary when encountered on their own.

No new special characters will ever be defined. All extensions to the regular expression syntax are made by defining new two-character constructs that begin with '`\`'.

`\|` specifies an alternative. Two regular expressions *a* and *b* with '`\|`' in between form an expression that matches anything that either *a* or *b* will match.

Thus, '`foo\|bar`' matches either '`foo`' or '`bar`' but no other string.

'`\|`' applies to the largest possible surrounding expressions. Only a surrounding '`\( ... \)`' grouping can limit the grouping power of '`\|`'.

Full backtracking capability exists to handle multiple uses of '`\|`'.

`\( ... \)`

is a grouping construct that serves three purposes:

1. To enclose a set of '`\|`' alternatives for other operations. Thus, '`\(foo\|bar\)x`' matches either '`foox`' or '`barx`'.
2. To enclose a complicated expression for the postfix '`*`' to operate on. Thus, '`ba\(na\)*`' matches '`bananana`', etc., with any (zero or more) number of '`na`' strings.
3. To mark a matched substring for future reference.

This last application is not a consequence of the idea of a parenthetical grouping; it is a separate feature which happens to be assigned as a second meaning to the same '`\( ... \)`' construct because there is no conflict in practice between the two meanings. Here is an explanation of this feature:

`\`*digit*
: after the end of a '`\( ... \)`' construct, the matcher remembers the beginning and end of the text matched by that construct. Then, later on in the regular expression, you can use '`\`' followed by *digit* to mean "match the same text matched the *digit*'th time by the '`\( ... \)`' construct."

  The strings matching the first nine '`\( ... \)`' constructs appearing in a regular expression are assigned numbers 1 through 9 in order of their beginnings. '`\1`' through '`\9`' may be used to refer to the text matched by the corresponding '`\( ... \)`' construct.

  For example, '`\(.*\)\1`' matches any string that is composed of two identical halves. The '`\(.*\)`' matches the first half, which may be anything, but the '`\1`' that follows must match the same exact text.

`` \` ``
: matches the empty string, but only if it is at the beginning of the buffer.

`\'`
: matches the empty string, but only if it is at the end of the buffer.

`\b`
: matches the empty string, but only if it is at the beginning or end of a word. Thus, '`\bfoo\b`' matches any occurrence of '`foo`' as a separate word. '`\bball\(s\|\)\b`' matches '`ball`' or '`balls`' as a separate word.

`\B`
: matches the empty string, provided it is *not* at the beginning or end of a word.

`\<`
: matches the empty string, provided it is at the beginning of a word.

`\>`
: matches the empty string, provided it is at the end of a word.

`\w`
: matches any word-constituent character. The editor syntax table determines which characters these are.

`\W`
: matches any character that is not a word-constituent.

`\s`*code*
: matches any character whose syntax is *code*. *code* is a letter which represents a syntax code: thus, '`w`' for word constituent, '`-`' for whitespace, '`(`' for open-parenthesis, etc. See section 29.5 [Syntax], page 169.

`\S`*code*
: matches any character whose syntax is not *code*.

14.6 Searching and Case

All sorts of searches in Emacs normally ignore the case of the text they are searching through; if you specify searching for 'FOO', then 'Foo' and 'foo' are also considered a match. If you do not want this feature, set the variable case-fold-search to nil. This variable has separate values in all individual buffers; in a new buffer, its value is initialized from default-case-fold-search. See section 29.2 [Variables], page 159.

14.7 Replacement Commands

Global search-and-replace operations are not needed as often in Emacs as they are in other editors, but they are available. In addition to the simple replace-string command which is like that found in most editors, there is a query-replace command which asks you, for each occurrence of the pattern, whether to replace it.

The replace commands all replace one string (or regexp) with one replacement string. It is possible to perform several replacements in parallel using the command expand-region-abbrevs. See section 25.2 [Controlling Expansion], page 140.

14.7.1 Unconditional Replacement

`M-x replace-string`
: Replace every occurrence of *string* with *newstring*.

`M-x replace-regexp`
: Replace every match for *regexp* with *newstring*.

To replace every instance of 'foo' after point with 'bar', use the command `M-x replace-string` with the two arguments 'foo' and 'bar'. Replacement occurs only after point, so if you want to cover the whole buffer you must go to the beginning first. All occurrences up to the end of the buffer are replaced; to limit replacement to part of the buffer, narrow to that part of the buffer before doing the replacement.

When replace-string exits, point is left at the last occurrence replaced. The value of point when the replace-string command was issued is remembered on the mark ring; `C-u C-SPC` moves back there.

replace-string replaces exact matches for a single string. The similar command replace-regexp replaces any match for a specified pattern.

In replace-regexp, the *newstring* need not be constant. It can refer to all or part of what is matched by the *regexp*. '\&' in *newstring* is replaced by the entire text being replaced. '*d*' in *newstring*, where *d* is a digit, is replaced by whatever matched the *d*'th parenthesized grouping in *regexp*. For example,

```
M-x replace-regexp RET c[ad]+r RET \&-safe RET
```

would replace (for example) '`cadr`' with '`cadr-safe`' and '`cddr`' with '`cddr-safe`'.

```
M-x replace-regexp RET \(c[ad]+r\)-safe RET \1 RET
```

would perform exactly the opposite replacements.

A numeric argument to either of the replace- commands restricts replacement to matches that are surrounded by word boundaries.

If the arguments to replace-string are in lower case, it preserves case when it makes a replacement. Thus, the command

```
M-x replace-string RET foo RET bar RET
```

replaces a lower case '`foo`' with a lower case '`bar`', '`FOO`' with '`BAR`', and '`Foo`' with '`Bar`'. If upper case letters are used in the second argument, they remain upper case every time that argument is inserted. If upper case letters are used in the first argument, the second argument is always substituted exactly as given, with no case conversion. Likewise, if the variable case-replace is set to nil, replacement is done without case conversion. If case-fold-search is set to nil, case is significant in matching occurrences of '`foo`' to replace; also, case conversion of the replacement string is not done.

14.7.2 Query Replace

`M-%`
`M-x query-replace`
: Replace some occurrences of one string with another string.

`M-x query-replace-regexp`
: Replace some matches for a regexp with a specified string.

If you want to change only some of the occurrences of '`foo`' to '`bar`', not all of them, then you cannot use an ordinary **replace-string**. Instead, use `M-%` (**query-replace**). This command finds occurrences of '`foo`' one by one, displays each occurrence and asks you whether to replace it. A numeric argument to **query-replace** tells it to consider only occurrences of '`foo`' that are bounded by word-delimiter characters.

Aside from querying, **query-replace** works just like **replace-string**, and **query-replace-regexp** works just like **replace-regexp**.

The things you can type when you are shown an occurrence of '`foo`' are:

`SPC`
: to replace the '`foo`' with '`bar`'. This preserves case, just like **replace-string**, provided **case-replace** is non-**nil**, as it normally is.

`DEL`
: to skip to the next '`foo`' without replacing this one.

`,`
: to replace this '`foo`' and display the result. You are then asked for another input character, except that since the replacement has already been made, `DEL` and `SPC` are equivalent.

`ESC`
: to exit without doing any more replacements.

`.`
: to replace this '`foo`' and then exit.

`!`
: to replace all remaining occurrences of '`foo`' without asking again.

`^`
: to go back to the location of the previous '`foo`' (or what used to be a '`foo`'), in case changed it by mistake. This works by popping the mark ring. Only one `^` is allowed, because only one previous replacement location is kept during **query-replace**.

`C-r`
: to enter a recursive editing level, in case the '`foo`' needs to be edited rather than just replaced with a '`bar`'. When you are done, exit the recursive editing level with `C-M-c` and the next '`foo`' will be displayed. See section 27.1 [Recursive Edit], page 149.

`C-w`
: to delete the '`foo`', and then start editing the buffer. When you are finished editing whatever is to replace the '`foo`', exit the recursive editing level with `C-M-c` and the next '`foo`' will be displayed.

`C-l`
: to redisplay the screen and then give another answer.

If you type any other character, the **query-replace** is exited, and the character executed as a command. To restart the **query-replace**, use `C-x ESC`, which repeats the **query-replace** because it used the minibuffer to read its arguments. See section 6.4 [Repetition], page 29.

14.8 Other Search-and-Loop Commands

Here are some other commands that find matches for a regular expression. They all operate from point to the end of the buffer.

`M-x list-matching-lines`
: Print each line that follows point and contains a match for the specified regexp. A numeric argument specifies the number of context lines to print before and after each matching line; the default is none.

`M-x count-occurrences`
: Print the number of matches following point for the specified regexp.

`M-x delete-non-matching-lines`
: Delete each line that follows point and does not contain a match for the specified regexp.

`M-x delete-matching-lines`
: Delete each line that follows point and contains a match for the specified regexp.

15. Commands for Fixing Typos

In this chapter we describe the commands that are especially useful for the times when you catch a mistake in your text just after you have made it, or change your mind while composing text on line.

`DEL`	Delete last character.
`M-DEL`	Kill last word.
`C-x DEL`	Kill to beginning of sentence.
`C-t`	Transpose two characters.
`M-t`	Transpose two words.
`C-M-t`	Transpose two balanced expressions.
`C-x C-t`	Transpose two lines.
`M-- M-l`	Convert last word to lower case. `Meta--` is Meta-minus!
`M-- M-u`	Convert last word to all upper case.
`M-- M-c`	Convert last word to lower case with capital initial.
`M-$`	Check and correct spelling of word.
`M-x spell-buffer`	Check and correct spelling of each word in the buffer.
`M-x spell-region`	Check and correct spelling of each word in the region.
`M-x spell-string`	Check spelling of specified word.

15.1 Killing Your Mistakes

The `DEL` character (delete-backward-char) is the most important correction command. When used among graphic (self-inserting) characters, it can be thought of as canceling the last character typed.

When your mistake is longer than a couple of characters, it might be more convenient to use `M-DEL` or `C-x DEL`. `M-DEL` kills back to the start of the last word, and `C-x DEL` kills back to the start of the last sentence. `C-x DEL` is particularly useful when you are thinking of what to write as you type it, in case you change your mind about phrasing. `M-DEL` and `C-x DEL` save the killed text for `C-y` and `M-y` to retrieve. See section 10.2 [Yanking], page 43.

M-DEL is often useful even when you have typed only a few characters wrong, if you know you are confused in your typing and aren't sure exactly what you typed. At such a time, you cannot correct with DEL except by looking at the screen to see what you did. It requires less thought to kill the whole word and start over again.

15.2 Transposing Text

The common error of transposing two characters can be fixed, when they are adjacent, with the C-t command (transpose-chars). Normally, C-t transposes the two characters on either side of point. When given at the end of a line, rather than transposing the last character of the line with the newline, which would be useless, C-t transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a C-t. If you don't catch it so fast, you must move the cursor back to between the two transposed characters. If you transposed a space with the last character of the word before it, the word motion commands are a good way of getting there. Otherwise, a reverse search (C-r) is often the best way. See chapter 14 [Search], page 55.

Meta-t (transpose-words) transposes the word before point with the word after point. It moves point forward over a word, dragging the word preceding or containing point forward as well. The punctuation characters between the words do not move. For example, 'FOO, BAR' transposes into 'BAR, FOO' rather than 'BAR FOO,'.

C-M-t (transpose-sexps) is a similar command for transposing two expressions (see section 23.2 [Lists], page 112), and C-x C-t (transpose-lines) exchanges lines. They work like M-t except in determining the division of the text into syntactic units.

A numeric argument to a transpose command serves as a repeat count: it tells the transpose command to move the character (word, sexp, line) before or containing point across several other characters (words, sexps, lines). For example, C-u 3 C-t moves the character before point forward across three other characters. This is equivalent to repeating C-t three times. C-u - 4 M-t moves the word before point backward across four words. C-u - C-M-t would cancel the effect of plain C-M-t.

A numeric argument of zero is assigned a special meaning (because otherwise a command with a repeat count of zero would do nothing): to transpose the character (word, sexp, line) before point with the one before the mark.

15.3 Case Conversion

A very common error is to type words in the wrong case. Because of this, the word case-conversion commands `M-l`, `M-u` and `M-c` have a special feature when used with a negative argument: they do not move the cursor. As soon as you see you have mistyped the last word, you can simply case-convert it and go on typing. See section 22.7 [Case], page 108.

15.4 Checking and Correcting Spelling

To check the spelling of the word before point, and optionally correct it as well, use the command `M-$` (spell-word). This command runs an inferior process containing the spell program to see whether the word is correct English. If it is not, it asks you to edit the word (in the minibuffer) into a corrected spelling, and then does a query-replace to substitute the corrected spelling for the old one throughout the buffer.

If you exit the minibuffer without altering the original spelling, it means you do not want to do anything to that word. Then the query-replace is not done.

`M-x spell-buffer` checks each word in the buffer the same way that spell-word does, doing a query-replace if appropriate for every incorrect word.

`M-x spell-region` is similar but operates only on the region, not the entire buffer.

`M-x spell-string` reads a string as an argument and checks whether that is a correctly spelled English word. It prints in the echo area a message giving the answer.

16. File Handling

The basic unit of stored data is the file. Each program, each paper, lives usually in its own file. To edit a program or paper, you must tell Emacs to examine the file and prepare a buffer containing a copy of the file's text. This is called *visiting* the file. Editing commands apply directly to text in the buffer; that is, to the copy inside Emacs. Your changes only appear in the file itself when you *save* the buffer back into the file.

In addition to visiting and saving files, Emacs can delete, copy, rename, and append to files, and operate on file directories.

16.1 File Names

Most Emacs commands that operate on a file require you to specify the file name. (Saving and reverting are exceptions; the buffer knows which file name to use for them.) File names are specified using the minibuffer (see chapter 6 [Minibuffer], page 25). *Completion* is available, to make it easier to specify long file names. See section 6.3 [Completion], page 27.

There is always a *default file name* which will be used if you type just RET, entering an empty argument. Normally the default file name is the name of the file visited in the current buffer; this makes it easy to operate on that file with any of the Emacs file commands.

Each buffer has a default directory, normally the same as the directory of the file visited in that buffer. When Emacs reads a file name, if you do not specify a directory, the default directory is used. If you specify a directory in a relative fashion, with a name that does not start with a slash, it is interpreted with respect to the default directory. The default directory is kept in the variable default-directory, which has a separate value in every buffer.

For example, if the default file name is /u/rms/gnu/gnu.tasks then the default directory is /u/rms/gnu/. If you type just foo, which does not specify a directory, it is short for /u/rms/gnu/foo. ../.login would stand for /u/rms/.login. new/foo would stand for /u/rms/gnu/new/foo.

The default directory actually appears initially in the minibuffer when the file name is read. This serves two purposes: it shows you what the default is, so that you can type a relative file name and know with certainty what it will mean, and it allows you to edit the default to specify a different directory.

Note that it is legitimate to type an absolute file name after you enter the minibuffer, ignoring the presence of the default directory name as part of the text. The final minibuffer contents may look invalid, but that is not so. See section 6.1 [Minibuffer File], page 25.

The command `M-x pwd` prints the current buffer's default directory, and the command `M-x cd` sets it (to a value read using the minibuffer). A buffer's default directory changes only when the `cd` command is used. A file-visiting buffer's default directory is initialized to the directory of the file that is visited there. If a buffer is made randomly with `C-x b`, its default directory is copied from that of the buffer that was current at the time.

16.2 Visiting Files

`C-x C-f`	Visit a file.
`C-x C-v`	Visit a different file instead of the one visited last.
`C-x 4 C-f`	Visit a file, in another window. Don't change this window.

Visiting a file means copying its contents into Emacs where you can edit them. Emacs makes a new buffer for each file that you visit. We say that the buffer is visiting the file that it was created to hold. Emacs constructs the buffer name from the file name by throwing away the directory, keeping just the name proper. For example, a file named /usr/rms/emacs.tex would get a buffer named '`emacs.tex`'. If there is already a buffer with that name, a unique name is constructed by appending '<2>', '<3>', or so on, using the lowest number that makes a name that is not already in use. If the variable ask-about-buffer-names is non-nil, the user is asked what buffer name to use; this takes the place of automatic uniquization.

Since the current buffer name always appears in the mode line, you can tell instantly from it which file you are editing.

The changes you make with Emacs are made in the Emacs buffer. They do not take effect in the file that you visited, or any place permanent, until you *save* the buffer. Saving the buffer means that Emacs writes the current contents of the buffer into its visited file. See section 16.3 [Saving], page 73.

If a buffer contains changes that have not been saved, the buffer is said to be *modified*. This is important because it implies that some changes will be lost if the buffer is not saved. The mode line displays two stars near the left margin if the current buffer is modified.

To visit a file, use the command C-x C-f (find-file). Follow the command with the name of the file you wish to visit, terminated by a RET.

The file name is read using the minibuffer (see chapter 6 [Minibuffer], page 25), with defaulting and completion in the standard manner (see section 16.1 [File Names], page 71). While in the minibuffer, you can abort C-x C-f by typing C-g.

Your confirmation that C-x C-f has completed successfully is the appearance of new text on the screen and a new buffer name in the mode line. If the specified file does not exist and could not be created, or cannot be read, then an error results. The error message is printed in the echo area, and includes the file name which Emacs was trying to visit.

If you visit a file that is already in Emacs, C-x C-f does not make another copy. It selects the existing buffer containing that file. However, before doing so, it checks that the file itself has not changed since you visited or saved it last. If the file has changed, a warning message is printed. See section 16.4.1 [Simultaneous Editing], page 76.

What if you want to create a file? Just visit it. Emacs prints '(New File)' in the echo area, but in other respects behaves as if you had visited an existing empty file. If you make any changes and save them, the file is created.

If you visit a nonexistent file unintentionally (because you typed the wrong file name), use the C-x C-v (find-alternate-file) command to visit the file you wanted. C-x C-v is similar to C-x C-f, but it kills the current buffer (after first offering to save it if it is modified).

If the file you specify is actually a directory, Dired is called on that directory (see chapter 17 [Dired], page 81). This can be inhibited by setting the variable find-file-run-dired to nil; then it is an error to try to visit a directory.

C-x 4 f (find-file-other-window) is like C-x C-f except that the buffer containing the specified file is selected in another window. The window that was selected before C-x 4 f continues to show the same buffer it was already showing. If this command is used when only one window is being displayed, that window is split in two, with one window showing the same before as before, and the other one showing the newly requested buffer.

16.3 Saving Files

Saving a buffer in Emacs means writing its contents back into the file that was visited in the buffer.

`C-x C-s`	Save the current buffer in its visited file.
`C-x s`	Save any or all buffers in their visited files.
`M-~`	Forget that the current buffer has been changed.
`C-x C-w`	Save the current buffer in a specified file, and record that file as the one visited in the buffer.
`M-x set-visited-file-name`	Mark the current buffer as visiting a specified file.

When you wish to save the file and make your changes permanent, type `C-x C-s` (save-buffer). After saving is finished, `C-x C-s` prints a message such as

```
Wrote /u/rms/gnu/gnu.tasks
```

If the selected buffer is not modified (no changes have been made in it since the buffer was created or last saved), saving is not really done, because it would be redundant. Instead, `C-x C-s` prints a message in the echo area saying

```
(No changes need to be written)
```

The command `C-x s` (save-some-buffers can save any or all modified buffers. First it asks, for each modified buffer, whether to save it. These questions appear as typeout, overlying the buffer text, and should be answered with `Y` or `N`. After all questions have been asked, the buffers you have approved are all saved.

If you have changed a buffer and do not want the changes to be saved, you should take some action to prevent it. Otherwise, each time you use save-some-buffers you are liable to save it by mistake. One thing you can do is type `M-~` (not-modified), which clears out the indication that the buffer is modified. If you do this, none of the save commands will believe that the buffer needs to be saved. (If we take '`~`' to mean 'not', then `Meta-~` is 'not', metafied.) You could also use set-visited-file-name (see below) to mark the buffer as visiting a different file name, one which is not in use for anything important. Alternatively, you can undo all the changes made since the file was visited or saved, by reading the text from the file again. This is called *reverting*. See section 16.5 [Reverting], page 77. You could also undo all the changes by repeating the undo command `C-x u` until you have undone all the changes; but this only works if you have not made more changes than the undo mechanism can remember.

`M-x set-visited-file-name` alters the name of the file that the current buffer is visiting. It reads the new file name using the minibuffer. It can be used on a buffer that is not visiting a file, too. The buffer's name is changed to correspond to the file it is now visiting in the usual fashion (unless the new name is in use already for some other buffer; in that case, the buffer name is not changed). set-visited-file-name does not save the buffer in the newly visited file; it just alters the records inside Emacs so that, if you save the buffer, it will be saved in that file. It also marks the buffer as "modified" so that `C-x C-s` *will* save.

If you wish to mark the buffer as visiting different file and save it right away, use `C-x C-w` (write-file). It is precisely equivalent to set-visited-file-name followed by `C-x C-s`. `C-x C-s` used on a buffer that is not visiting with a file has the same effect as `C-x C-w`; that is, it reads a file name, marks the buffer as visiting that file, and saves it there. The default file name in a buffer that is not visiting a file is made by combining the buffer name (as the name component of the pathname) with the other components taken from the last file-visiting buffer that was current.

If Emacs is about to save a file and sees that the date of the latest version on disk does not match what Emacs last read or wrote, Emacs notifies you of this fact, because it probably indicates a problem caused by simultaneous editing and requires your immediate attention. See section 16.4.1 [Interlocking], page 76.

If the variable require-final-newline is non-nil, Emacs puts a newline at the end of any file that doesn't already end in one, every time a file is saved or written.

16.4 Backup Files

Because Unix does not provide version numbers in file names, rewriting a file in Unix automatically destroys all record of what the file used to contain. Thus, saving a file from Emacs throws away the old contents of the file–or it would, except that Emacs carefully copies the old contents to another file, called the *backup* file, before actually saving. The backup file's name is constructed by appending '~' to the file name being edited; thus, the backup file for eval.c would be eval.c~.

Emacs makes a backup for a file only the first time the file is saved in each editing session. No matter how many times you save a file, its backup file continues to contain the contents from before the current editing session.

Backup files can be made by copying the old file or by renaming it. This makes a difference when the old file has multiple names. If the old file is renamed into the backup file, then the alternate names become names for the backup file. If the old file is copied instead, then the alternate names remain names for the file that you are editing, and the contents accessed by those names will be the new contents.

The choice of renaming or copying is controlled by two variables. Normally, renaming is done. If the variable **backup-by-copying** is non-**nil**, copying is used. If the variable **backup-by-copying-when-linked** is non-**nil**, then copying is done for files that have multiple names, but renaming is done when the file being edited has only one name. (For files with only one name, the major difference between renaming and copying is that renaming is faster.)

If the variable **make-backup-files** is set to **nil**, backup files are not written at all.

16.4.1 Protection against Simultaneous Editing

Simultaneous editing occurs when two users visit the same file, both make changes, and then both save them. If nobody were informed that this was happening, whichever user saved first would later find that his changes were lost. Emacs cannot prevent users from editing simultaneously, but it always warns at least one of the users (the one who saves last) that he is about to lose. If he takes the proper corrective action at this point, he can prevent a problem.

Every time Emacs saves a buffer, it first checks the last-modification-date of the existing file on disk to see that it has not changed since the file was last visited or saved. If the date does not match, it implies that changes were made in the file in some other way, and these changes are about to be lost if Emacs actually does save. To prevent this, Emacs prints a warning message and asks for confirmation before saving. Occasionally you will know why the file was changed and know that it does not matter; then you can answer 'yes' and proceed. Otherwise, you should cancel the save with `C-g` and investigate the situation.

The first thing you should do when notified of simultaneous editing is list the directory with `C-u C-x C-d` (see section 16.7 [Directory Listing], page 79). This will show the file's current author. You should attempt to contact him to warn him not to continue editing. Often the next step is to save the contents of your Emacs buffer under a different name, and use **diff** to compare the two files.

Simultaneous editing checks are also made when you visit with `C-x C-f` a file that is already visited. This is not strictly necessary, but it can cause you to find out about the problem earlier, when perhaps correction takes less work.

16.5 Reverting a Buffer

If you have made extensive changes to a file and then change your mind about them, you can get rid of them by reading in the previous version of the file. To do this, use `M-x revert-buffer`, which operates on the current buffer. Since this is a very dangerous thing to do, you must confirm it with 'yes'.

If the current buffer has been auto-saved more recently than it has been saved for real, **revert-buffer** offers to read the auto save file instead of the visited file. This question comes before the usual request for confirmation, and demands `y` or `n` as an answer. If you have started to type **yes** for confirmation without realizing that the other question was going to be asked, the `y` will answer that question, but the **es** will not be valid confirmation. So you will have a chance to cancel the operation with **C-g** and try it again with the answers that you really intend.

revert-buffer keeps point at the same distance (measured in characters) from the beginning of the file. If the file was edited only slightly, you will be at approximately the same piece of text after reverting as before. If you have made drastic changes, the same value of point in the old file may address a totally different piece of text.

A reverted buffer is marked "not modified" until another change is made.

Some kinds of buffers whose contents reflect data bases other than files, such as Dired buffers, can also be reverted. For them, reverting means recalculating their contents from the appropriate data base. Buffers created randomly with `C-x b` cannot be reverted; **revert-buffer** reports an error when asked to do so.

16.6 Auto Saving: Protection Against Disasters

Emacs saves all the visited files from time to time (based on counting your keystrokes) without being asked. This is called *auto-saving*. It prevents you from losing more than a limited amount of work if the system crashes.

Auto-saving does not normally save in the files that you visited, because it can be very undesirable to save a program that is in an inconsistent state because you have made half of a planned change. Instead, auto-saving is done in a different file called the *auto-save file*, and the visited file is changed only when you request saving explicitly (such as with `C-x C-s`). If you want auto-saving to be done in the visited file, set the variable **auto-save-visited-file-name** to be non-**nil**. The file name to be used for auto-saving in a buffer is calculated when auto-saving is turned on in that buffer, based on the variable values in effect at that time.

Normally, the auto-save file name is made by appending '#' to the front of the visited file. Thus, a buffer visiting file foo.c would be auto-saved in a file #foo.c. Most buffers that are not visiting files are auto-saved only if you request it explicitly; when they are auto-saved, the auto-save file name is made by appending '#%' to the buffer name. For example, the *mail* buffer in which you compose messages to be sent is auto-saved in a file named #%*mail*. Auto-save file names are made this way unless you reprogram parts of Emacs to do something different.

Each time you visit a file, no matter how, auto saving is turned on for that file if the variable **auto-save-default** is non-**nil**. The default for this variable is **t**, so auto-saving is the usual practice for file-visiting buffers. Auto-saving can be turned on or off for any existing buffer with the command `M-x auto-save-mode`. Like other minor mode commands, `M-x auto-save-mode` turns auto-saving on with a positive argument, off with a zero or negative argument; with no argument, it toggles.

Emacs does auto-saving every so often, based on counting how many characters you have typed since the last time auto-saving was done. The variable **auto-save-interval** specifies how many characters there are between auto-saves. By default, it is 300. Emacs also auto-saves whenever you call the function **do-auto-save**.

Emacs also does auto-saving whenever it gets a fatal error. This includes killing the Emacs job with a shell command such as **kill %emacs**, or disconnecting a phone line or network connection.

When Emacs determines that it is time for auto-saving, each buffer is considered, and is auto-saved if auto-saving is turned on for it and it has been changed since the last time it was auto-saved. If any auto-saving is done, the message '`Auto-saving...`' is displayed in the echo area until auto-saving is finished. Errors occurring during auto-saving are trapped so that they do not interfere with the execution of commands you have been typing.

You may wish to have a buffer's auto-save file deleted when you save the buffer in its visited file. To request this, set the variable delete-auto-save-files non-nil. The variable is normally nil, so that auto-save files remain until explicitly deleted.

The way to use the contents of an auto save file to recover from a loss of data is to make a buffer with `C-x b`, read the auto-save file into it with `M-x insert-file`, and then write it into the desired file name with `C-x C-w`. For example, to recover file foo.c from its auto-save file #foo.c, do:

```
C-x b temp RET
M-x insert-file RET #foo.c RET
C-x C-w foo.c RET
```

16.7 Listing a File Directory

Files are classified into *directories*. A *directory listing* is a list of all the files in a directory. Emacs provides directory listings in brief format (file names only) and verbose format (sizes, dates, and authors included).

`C-x C-d` *dir-or-pattern*
: Print a brief directory listing.

`C-u C-x C-d` *dir-or-pattern*
: Print a verbose directory listing.

The command to print a directory listing is `C-x C-d` (list-directory). It reads using the minibuffer a file name which is either a directory to be listed or a wildcard-containing pattern for the files to be listed. For example,

```
C-x C-d /u2/emacs/etc RET
```

lists all the files in directory /u2/emacs/etc. An example of specifying a file name pattern is

```
C-x C-d /u2/emacs/src/*.c RET
```

In accepting either form of argument, `C-x C-d` resembles the Unix directory-listing command ls.

Normally, `C-x C-d` prints a brief directory listing containing just file names. A numeric argument (regardless of value) tells it to print a verbose listing (like ls -l).

The text of a directory listing is obtained by running ls in an inferior process. Two Emacs variables control the switches passed to ls: list-directory-brief-switches is a string giving the switches to use in brief listings ("-CF" by default), and list-directory-verbose-switches is a string giving the switches to use in a verbose listing ("-l" by default).

17. Dired, the Directory Editor

Dired makes it easy to delete or visit many of the files in a single directory at once. It makes an Emacs buffer containing a listing of the directory. You can use the normal Emacs commands to move around in this buffer, and special Dired commands to operate on the files.

To invoke dired, do `C-x d` or `M-x dired`. The command reads a directory name or wildcard file name pattern as a minibuffer argument just like the **list-directory** command, `C-x C-d`. Where **dired** differs from **list-directory** is in naming the buffer after the directory name or the wildcard pattern used for the listing, and putting the buffer into Dired mode so that the special commands of Dired are available in it. The variable **dired-listing-switches** is a string used as an argument to **ls** in making the directory; this string *must* contain '`-l`'.

Once the D:red buffer exists, you can switch freely between it and other Emacs buffers. Whenever the Dired buffer is selected, certain special commands are provided that operate on files that are listed. The Dired buffer is "read-only", and inserting text in it is not useful, so ordinary printing characters such as `d` and `x` are used for Dired commands. Most Dired commands operate on the file described by the line that point is on. Some commands perform operations immediately; others "mark" the file to be operated on later.

Most Dired commands that operate on the current line's file also treat a numeric argument a repeat count, meaning to apply to the files of the next few lines. A negative argument means to operate on the files of the preceding lines, and leave point on the first of those lines.

All the usual Emacs cursor motion commands are available in Dired buffers. Some special purpose commands are also provided.

For extra convenience, `SPC` in Dired is a command similar to `C-N`. Moving down a line is done so often in Dired that it deserves to be easy to type. `DEL` is often useful simply for moving up.

17.1 Deleting Files with Dired

The primary use of Dired is to mark files for deletion and then delete them.

`d`	Mark this file for deletion.
`u`	Remove deletion-mark on this line.
`DEL`	Remove deletion-mark on previous line, moving point to that line.

x Delete the files that are marked for deletion.

Mark all auto-save files (files whose names start with '#') for deletion (see section 16.6 [Auto Save], page 77).

~ Mark all backup files (files whose names end with '~') for deletion (see section 16.4 [Backup], page 75).

You can mark a file for deletion by moving to the line describing the file and typing d or C-d. The deletion mark is visible as a 'D' at the beginning of the line. Point is moved to the beginning of the next line, so that repeated d commands mark successive files.

The files are marked for deletion rather than deleted immediately to avoid the danger of deleting a file accidentally. Until you direct Dired to delete the marked files, you can remove deletion marks using the commands u and DEL. u works just like d, but removes marks rather than making marks. DEL moves upward, removing marks; it is like u with numeric argument automatically negated.

To delete the marked files, type x. This command first displays a list of all the file names marked for deletion, and requests confirmation with 'yes'. Once you confirm, all the marked files are deleted, and their lines are deleted from the text of the Dired buffer. The shortened Dired buffer remains selected. If you answer 'no' or quit with C-g, you return imediately to Dired, with the deletion marks still present and no files actually deleted.

The # and ~ commands marks many files for deletion, based on their names. # marks for deletion all files that appear to have been made by auto-saving (that is, files whose names begin with '#'). ~ marks for deletion all files that appear to have been made as backups for files that were edited (that is, files whose names end with '~'). These commands are useful precisely because they do not actually delete any files; you can remove the deletion marks from any marked files that you really wish to keep.

17.2 Immediate File Operations in Dired

Some file operations in Dired take place immediately when they are requested.

c Copies the file described on the current line. You must supply a file name to copy to, using the minibuffer.

f Visits the file described on the current line. It is just like typing C-x C-f and supplying that file name. If the file on this line is a subdirectory, f actually causes Dired to be invoked on that subdirectory. See section 16.2 [Visiting], page 72.

o	Like f, but uses another window to display the file's buffer. The Dired buffer remains visible in the first window. This is like using `C-x 4 C-f` to visit the file. See chapter 19 [Windows], page 91.
r	Renames the file described on the current line. You must supply a file name to rename to, using the minibuffer.
v	Views the file described on this line using `M-x view-file`. Viewing a file is like visiting it, but is slanted toward moving around in the file conveniently and does not allow changing the file. See section 17.3 [Other File Operations], page 83.

17.3 Miscellaneous File Operations

Emacs has extended commands for performing many other operations on files.

`M-x view-file` allows you to scan or read a file by sequential screenfuls. It reads a file name argument using the minibuffer. After reading the file into an Emacs buffer, view-file reads and displays one window full. You can then type SPC to scroll forward one window full, or DEL. Various other commands are provided for moving around in the file, but none for changing it. To exit from viewing, type `C-c`.

`M-x insert-file` inserts the contents of the specified file into the current buffer at point, leaving point unchanged before the contents and an inactive mark after them. See chapter 9 [Mark], page 37.

`M-x write-region` is the inverse of `M-x insert-file`; it copies the contents of the region into the specified file. `M-x append-to-file` adds the text of the region to the end of the specified file.

`M-x delete-file` deletes the specified file, like the `rm` command in the shell. If you are deleting many files in one directory, it may be more convenient to use Dired (see chapter 17 [Dired], page 81).

`M-x rename-file` reads two file names *old* and *new* using the minibuffer, then renames file *old* as *new*. If a file named *new* already exists, you must confirm with 'yes' or renaming is not done. The similar command `M-x add-name-to-file` is used to add an additional name to an existing file without removing its old name.

`M-x copy-file` works just like `M-x rename-file`, but copies the files *old* to the files *new*. Confirmation is required if *new* exists.

`M-x make-symbolic-link` reads two file names *old* and *linkname*, and then creates a symbolic link named *linkname* and pointing at *old*. The effect is that future attempts to open file *linkname* will refer to whatever file is named *old* at the time the opening is done, or will get an error if the name *old* is not in use at that time. Confirmation is required when creating the link if *linkname* is in use. Note that not all systems support symbolic links.

18. Using Multiple Buffers

The text you are editing in Emacs resides in an object called a *buffer*. Each time you visit a file, a buffer is created to hold the file's text. Each time you invoke Dired, a buffer is created to hold the directory listing. If you a message with `C-x m`, a buffer named *mail* is used to hold the text of the message. When you ask for a command's documentation, that appears in a buffer called *Help*.

At any time, one and only one buffer is *selected*. It is also called the *current buffer*. Often we say that a command operates on "the buffer" as if there were only one; but really this means that the command operates on the selected buffer (most commands do).

When Emacs makes multiple windows, each window has a chosen buffer which is displayed there, but at any time only one of the windows is selected and its chosen buffer is the selected buffer. Each window's mode line displays the name of the buffer that the window is displaying.

Each buffer has a name, which can be of any length, and you can select any buffer by giving its name. Most buffers are made by visiting files, and their names are derived from the files' names. But you can also create an empty buffer with any name you want. A newly started Emacs has a buffer named '`*scratch*`' which can be used for evaluating Lisp expressions in Emacs. The distinction between upper and lower case matters in buffer names.

Each buffer records individually what file it is visiting, whether it is modified, and what major mode and minor modes are in effect in it. Any Emacs variable can be made *local to* a particular buffer, meaning its value in that buffer can be different from the value in other buffers. See section 29.2 [Variables], page 159.

18.1 Creating and Selecting Buffers

`C-x b`	Select or create a buffer.
`C-x C-b`	List the existing buffers.

To select the buffer named *bufname*, type `C-x b` *bufname* `RET`. This is the command switch-to-buffer with argument *bufname*. Because completion is provided for buffer names, you can abbreviate the buffer name (see section 6.3 [Completion], page 27). An empty argument to `C-x b` specifies the most recently selected buffer that is not displayed in any window.

To print a list of all the buffers that exist, type `C-x C-b` (list-buffers). Each buffer's name, major mode and visited file are printed. '*' at the beginning of a line indicates the buffer is "modified". If several buffers are modified, it may be time to save some with `C-x s` (see section 16.3 [Saving], page 73). '%' indicates a read-only buffer. '.' marks the selected buffer. Here is an example of a buffer list:

```
 MR Buffer         Size  Mode           File
 -- ------         ----  ----           ----
.*  gmacs.tex      421336 Text          /u2/emacs/man/gmacs.tex
    *Help*         1287  Fundamental
    files.el       23076 Emacs-Lisp     /u2/emacs/lisp/files.el
  % RMAIL          64042 RMAIL          /u/rms/RMAIL
    emacs.tex      383402 Text          /u2/emacs/man/emacs.tex
 *% man            747   Dired
    net.emacs      343885 Fundamental   /u/rms/net.emacs
    fileio.c       27691 C              /u2/emacs/src/fileio.c
    NEWS           67340 Text           /u2/emacs/etc/NEWS
```

Note that the buffer *Help* was made by a help request; it is not visiting any file. The buffer man was made by Dired on the directory /u2/emacs/man.

Most buffers are created by visiting files, or by Emacs commands that want to display some text, but you can also create a buffer explicitly by typing `C-x b` *bufname* `RET`. This makes a new, empty buffer which is not visiting any file, and selects it for editing. Such buffers are used for making notes to yourself. If you try to save one, you are asked for the file name to use. The new buffer's major mode is determined by the value of default-major-mode (see chapter 20 [Major Modes], page 95.

Note that `C-x C-f`, and any other command for visiting a file, can also be used to switch buffers. See chapter 16 [Files], page 71.

18.2 Miscellaneous Buffer Operations

`C-x C-q` Toggle read-only status of buffer.

`M-x rename-buffer`
  Change the name of the current buffer.

`M-x view-buffer`
  Scroll through a buffer.

A buffer can be *read-only*, which means that commands to change its text are not allowed. Normally, read only buffers are made by subsystems such as Dired and Rmail that have special commands to operate on the text; a read-only buffer is also made if you visit a file that is protected so you cannot write it. If you wish to make changes in a read-only buffer, use the command `C-x C-q` (toggle-read-only). It makes a read-only buffer writable, and makes a writable buffer read-only. This works by setting the variable buffer-read-only, which has a local value in each buffer and makes the buffer read-only if its value is non-nil.

`M-x rename-buffer` changes the name of the current buffer. Specify the new name as a minibuffer argument. There is no default. If you specify a name that is in use for some other buffer, an error happens and no renaming is done.

`M-x view-buffer` is much like `M-x view-file` (see section 17.3 [Filadv], page 83) except that it examines an already existing Emacs buffer. View mode provides commands for scrolling through the buffer con.eniently but not for changing it. When you exit View mode, the value of point that resulted from your perusal remains in effect.

The commands `C-x a` (append-to-buffer) and `M-x insert-buffer` can be used to copy text from one buffer to another. See section 10.3 [Accumulating Text], page 46.

18.3 Killing Buffers

After you use Emacs for a while, you may accumulate a large number of buffers. You may then find it convenient to eliminate the ones you no longer need. There are several commands provided for doing this.

`C-x k` Kill a buffer, specified by name.

`M-x kill-some-buffers`
Offer to kill each buffer, one by one.

`C-x k` (kill-buffer) kills one buffer, whose name you specify in the minibuffer. The default, used if you type just RET in the minibuffer, is to kill the current buffer. If the current buffer is killed, another buffer is selected; a buffer that has been selected recently but does not appear in any window now is chosen to be selected. If the buffer being killed is modified (has unsaved editing) then you are asked to confirm with 'yes' before the buffer is killed.

The command `M-x kill-some-buffers` asks about each buffer, one by one. An answer of `Y` means to kill the buffer. Killing the current buffer or a buffer containing unsaved changes selects a new buffer or asks for confirmation just like kill-buffer.

18.4 Operating on Several Buffers

The *buffer-menu* facility is like a "Dired for buffers"; it allows you to request operations on various Emacs buffers by editing an Emacs buffer containing a list of them.

`M-x buffer-menu`
: Begin editing a buffer listing all Emacs buffers.

The command buffer-menu writes a list of all Emacs buffers into the buffer *Buffer List*, and selects that buffer in Buffer Menu mode. The buffer is read-only, and can only be changed through the special commands described in this section. Most of these commands are graphic characters. The usual Emacs cursor motion commands can be used in the *Buffer List* buffer. The following special commands apply to the buffer described on the current line.

k	Request to kill the buffer. The request shows as a 'K' on the line, before the buffer name. Requested kills take place when the x command is used.
s	Request to save the buffer. The request shows as an 'S' on the line. Requested saves take place when the x command is used. You may request both saving and killing for one buffer.
~	Mark buffer "unmodified". The command ~ does this, immediately when typed.
x	Perform previously requested kills and saves.

There are also special commands to use the buffer list to select another buffer, and to specify one or more other buffers for display in additional windows.

1	Select the buffer in a full-screen window. This command takes effect immediately.
2	Use two windows, with this buffer in one, and the previously selected buffer (aside from the buffer *Buffer List*) in the other.
q	Select this buffer, and also display in other windows any buffers previously marked with the m command. If there are no such buffers, this command is equivalent to 1.
m	Mark this buffer to be displayed in another window if the q command is used. The request shows as a '>' at the beginning of the line. The same buffer may not have both a kill request and a display request.

All the commands that put in marks to request operations later also move down a line, and accept a numeric argument as a repeat count.

The command u cancels any request marked for the current line, and moves down; DEL does so for the previous line, and moves up to it.

All that buffer-menu does directly is create and select a suitable buffer, and turn on Buffer Menu mode. Everything else described above is implemented by the special commands provided in Buffer Menu mode. One consequence of this is that you can switch from the buffer-menu buffer to another Emacs buffer, and edit there. You can reselect the buffer-menu buffer later, to perform the operations already requested, or you can kill it, or pay no further attention to it.

The only difference between buffer-menu and list-buffers is that buffer-menu selects the *Buffer List* buffer and list-buffers does not. If you run list-buffers (that is, type C-x C-b) and select the buffer list manually, you can use all of the commands described above.

19. Multiple Windows

Emacs can slit the screen into two or many windows, which can display parts of different buffers, or different parts of one buffer.

When multiple windows are being displayed, each window has an Emacs buffer designated for display in it. The same buffer may appear in more than one window; if it does, any changes in its text are displayed in all the windows where it appears. But the windows showing the same buffer can show different parts of it, because each window has its own value of point.

At any time, one of the windows is the *selected window*; the buffer this window is displaying is the current buffer. The terminal's cursor shows the location of point in this window. Each other window has a location of point as well, but since the terminal has only one cursor there is no way to show wher those locations are.

Commands to move point affect the value of point for the selected Emacs window only. They do not change the value of point in any other Emacs window, even one showing the same buffer. The same is true for commands such as `C-x b` to change the selected buffer in the selected window; they do not affect other windows at all. However, there are other commands such as `C-x 4 b` that select a different window and switch buffers in it. Also, all commands that display information in a window, including (for example) `C-h f` (describe-function) and `C-x C-b` (list-buffers), work by switching buffers in a nonselected window without affecting the selected window.

Each window has its own mode line, which displays the buffer name, modification status and major and minor modes of the buffer that is displayed in the window. See section 1.3 [Mode Line], page 7, for full details on the mode line.

`C-x 2`	Split the selected window into two windows, one above the other.
`C-x 5`	Split the selected window into two windows, side by side.
`C-x o`	Select another window (o, not zero).
`C-x 0`	Get rid of the selected window (and select some other window).
`C-x 1`	Get rid of all windows except the current one.
`C-x 4`	Prefix key for commands to select a buffer in various ways "in another window".
`C-x ^`	Make the selected window bigger, at the expense of the other(s).
`C-M-v`	Scroll the other window.

The command `C-x 2` (split-window-vertically) breaks the selected window into two windows, one above the other. Both windows start out displaying the same buffer, with the same value of point. By default the two windows each get half the height of the window that was split; a numeric argument specifies how many lines to give to the top window.

C-x 5 (split-window-horizontally) breaks the selected window into two side-by-side windows. A numeric argument specifies how many columns to give the one on the left. A line of vertical bars separates the two windows. Windows that are not the full width of the screen have mode lines, but they are truncated; also, they do not appear in inverse video. The Emacs display routines have not been told how to display a region of inverse video that is only part of a line on the screen.

When a window is less than the full width, text lines too long to fit are frequent. Continuing all those lines might be confusing. The variable truncate-partial-width-windows can be set non-nil to force truncation in all windows less than the full width of the screen, independent of the buffer being displayed and its value for truncate-lines. See section 4.2 [Continuation Lines], page 18.

Horizontal scrolling is often used in side-by-side windows. See chapter 13 [Display], page 53.

To select a different window, use C-x o (other-window). That is an o, for 'other', not a zero. When there are more than two windows, this command moves through all the windows in a cyclic order, generally top to bottom and left to right. From the rightmost and bottommost window, it goes back to the one at the upper left corner. A numeric argument means to move several steps in the cyclic order of windows. A negative argument moves around the cycle in the opposite order. When the minibuffer is active, the minibuffer is the last window in the cycle; you can switch from the minibuffer window to one of the other windows, and later switch back and finish supplying the minibuffer argument that is requested. See section 6.2 [Minibuffer Edit], page 26.

The usual scrolling commands (see chapter 13 [Display], page 53) apply to the selected window only, but there is one command to scroll the next window. C-M-v (scroll-other-window) scrolls the window that C-x o would select. The kind of scrolling done is the same as for C-v.

To delete a window, type C-x 0 (delete-window). The space it used to occupy is distributed among the other active windows (but not the minibuffer window, even if that is active at the time). Once a window is deleted, everything about it is forgotten; there is no automatic way to make another window showing the same contents.

C-x 1 (delete-other-windows) is more powerful than C-x 0; it deletes all the windows except the selected one (and the minibuffer); the selected window expands to use the whole screen except for the echo area.

To readjust the division of space among existing windows, use C-x ^ (enlarge-window). It makes the currently selected window get one line bigger, or as many lines as is specified with a numeric argument. With a negative argument, it makes the selected window smaller. `C-x }` (enlarge-window-horizontally) makes the selected window wider by the specified number of columns. The extra screen space given to a window comes from one of its neighbors, if that is possible; otherwise, all the competing windows are shrunk in the same proportion. If this makes any windows too small, those windows are deleted and their space is divided up. The minimum size is specified by the variables window-min-height and window-min-width.

`C-x 4` is a prefix key for commands that select another window (splitting the window if there is only one) and select a buffer in that window. Different `C-x 4` commands have different ways of finding the buffer to select.

`C-x 4 b` *bufname* RET
: Select buffer *bufname* in another window. This runs pop-to-window.

`C-x 4 f` *filename* RET
: Visit file *filename* and select its buffer in another window. This runs find-file-other-window.

`C-x 4 d` *directory* RET
: Select a Dired buffer for directory *directory* in another window. This runs dired-other-window.

`C-x 4 m`
: Start composing a mail message in another window. This runs mail-other-window, and its same-window version is `C-x m`. See section 27.2 [Mail], page 150.

`C-x 4 .`
: Find a tag in the current tag table in another window. This runs find-tag-other-window, the multiple-window variant of `M-.`. See section 23.10 [Tags], page 125.

20. Major Modes

Emacs has many different *major modes*, each of which customizes Emacs for editing text of a particular sort. The major modes are mutually exclusive, and each buffer has one major mode at any time. The mode line normally contains the name of the current major mode, in parentheses. See section 1.3 [Mode Line], page 7.

The least specialized major mode is called *Fundamental mode*. This mode has no mode-specific redefinitions or variable settings, so that each Emacs command behaves in its most general manner, and each option is in its default state. For editing any specific type of text, such as Lisp code or English text, you should switch to the appropriate major mode, such as Lisp mode or Text mode.

Selecting a major mode changes the meanings of a few keys to become more specifically adapted to the language being edited. The ones which are changed frequently are `TAB`, `DEL`, and `LFD`. In addition, the commands which handle comments use the mode to determine how comments are to be delimited. Many major modes redefine the syntactical properties of characters appearing in the buffer. See section 29.5 [Syntax], page 169.

The major modes fall into three major groups. Lisp mode (which has several variants), C mode and Muddle mode are for specific programming languages. Text mode is for editing English text. The remaining major modes are not intended for use on user's files; they are used in buffers created for specific purposes by Emacs, such as Dired mode for buffers made by Dired (see chapter 17 [Dired], page 81), and Mail mode for buffers made by `C-x m` (see section 27.2 [Mail], page 150), and Shell mode for buffers used for communicating with an inferior shell process (see section 28.1 [Shell], page 153).

Selecting a new major mode is done with an `M-x` command. From the name of a major mode, add `-mode` to get the name of a command function to select that mode. Thus, you can enter Lisp mode by executing `M-x lisp-mode`.

When you visit a file, Emacs usually chooses the right major mode based on the file's name. For example, files whose names end in .c are edited in C mode. The correspondence between file names and major mode is controlled by the variable auto-mode-alist. Its value is a list in which each element has the form

(*regexp* . *mode-function*)

For example, one element normally found in the list has the form `("\\.c$" . c-mode)`, and it is responsible for selecting C mode for files whose names end in .c.

You can specify which major mode should be used for editing a certain file by a special sort of text in the first nonblank line of the file. The mode name should appear in this line both preceded and followed by '-*-'. Other text may appear on the line as well. For example,

```
;-*-Lisp-*-
```

tells Emacs to use Lisp mode. Note how the semicolon is used to make Lisp treat this line as a comment. Such an explicit specification overrides any defaulting based on the file name.

Another format of mode specification is

```
-*-Mode: modename;-*-
```

which allows other things besides the major mode name to be specified. However, Emacs does not look for anything except the mode name.

When a file is visited that does not specify a major mode to use, or when a new buffer is created with `C-x b`, the major mode used is that specified by thef variable **default-major-mode**. Normally this value is the symbol **fundamental-mode**, which specifies Fundamental mode. If **default-major-mode** is **nil**, the major mode is taken from the previously selected buffer.

Most programming language major modes specify that only blank lines separate paragraphs. This is so that the paragraph commands remain useful. See section 22.4 [Paragraphs], page 104. They also cause Auto Fill mode to use the definition of TAB to indent the new lines it creates. This is because most lines in a program are usually indented. See chapter 21 [Indentation], page 97.

21. Indentation

TAB	Indent current line "appropriately" in a mode-dependent fashion.
LFD	Perform RET followed by TAB.
M-^	Merge two lines. This would cancel out the effect of LFD.
C-M-o	Split line at point; text on the line after point becomes a new line indented to the same column that it now starts in.
M-m	Move (forward or back) to the first nonblank character on the current line.
C-M-\	Indent several lines to same column.
C-x TAB	Shift block of lines rigidly right or left.
M-x indent-relative	Indent to an indentation point in the previous line.

Most programming languages have some indentation convention. For Lisp code, lines are indented according to their nesting in parentheses. The same general idea is used for C code, though many details are different.

Whatever the language, to indent a line, use the TAB command. Each major mode defines this command to perform the sort of indentation appropriate for the particular language. In Lisp mode, TAB aligns the line according to its depth in parentheses. No matter where in the line you are when you type TAB, it aligns the line as a whole. In C mode, TAB implements a subtle and sophisicated indentation style that knows about many aspects of C syntax.

In Text mode, TAB runs the command tab-to-tab-stop, which indents to the next tab stop column. You can set the tab stops with `M-x edit-tab-stops`.

If you just want to insert a tab character in the buffer, you can type `C-q TAB`.

To move over the indentation on a line, do `Meta-m` (back-to-indentation). This commands, given anywhere on a line, positions point at the first nonblank character on the line.

To insert an indented line before the current line, do `C-a C-o TAB`. To make an indented line after the current line, use `C-e LFD`.

`C-M-o` (split-line) moves the text from point to the end of the line vertically down, so that the current line becomes two lines. `C-M-o` first moves point forward over any spaces and tabs. Then it inserts after point a newline and enough indentation to reach the same column point is on. Point remains before the inserted newline; in this regard, C-M-o resembles C-o.

To join two lines cleanly, use the Meta-^ (delete-indentation) command to delete the indentation at the front of the current line, and the line boundary as well. They are replaced by a single space, or by no space if at the beginning of a line or before a ')' or after a '('. To delete just the indentation of a line, go to the beginning of the line and use Meta-\ (delete-horizontal-space), which deletes all spaces and tabs around the cursor.

There are also commands for changing the indentation of several lines at once. Control-Meta-\ (indent-region) gives each line which begins in the region the "usual" indentation by invoking TAB at the beginning of the line. A numeric argument specifies the indentation, and each line is shifted left or right so that it has exactly that much. C-x TAB (indent-rigidly) moves all of the lines in the region right by its argument (left, for negative arguments). The whole group of lines move rigidly sideways, which is how the command gets its name.

M-x indent-relative indents at point based on the previous line (actually, the previous nonempty line.) It inserts whitespace at point, moving point, until it is underneath an indentation point in the previous line. An indentation point is the end of a sequence of whitespace or the end of the line. If point is farther right than any indentation point in the previous line, the whitespace before point is deleted and the first indentation point then applicable is used. If no indentation point is applicable even then, tab-to-tab-stop is run.

indent-relative is the definition of TAB in Indented Text mode. See chapter 22 [Text], page 101.

21.1 Tab Stops

For typing in tables, you can use Text mode's definition of TAB, tab-to-tab-stop. This command inserts indentation before point, enough to reach the next tab stop column. If you are not in Text mode, this function can be found on M-i anyway.

The tab stops used by M-i can be set arbitrarily by the user. They are stored in a variable called tab-stop-list, as a list of column-numbers in increasing order.

The convenient way to set the tab stops using M-x edit-tab-stops, which creates and selects a buffer containing a description of the tab stop settings. You can edit this buffer to specify different tab stops, and then type C-x C-s to make those new tab stops take effect. In the tab stop buffer, C-x C-s runs the function edit-tab-stops-note-changes rather than its usual definition save-buffer. edit-tab-stops records which buffer was current when you invoked it, and stores the tab stops back in that buffer; normally all buffers share the same tab stops and changing them in one buffer affects all, but if you happen to make **tab-stop-list** local in one buffer then **edit-tab-stops** will edit the tab stops that are effective in the buffer that you invoke it for.

Here is what the text representing the tab stops looks like for ordinary tab stops every eight columns.

```
        :       :       :       :       :       :       :       :       :
0         1         2         3         4         5         6         7
0123456789012345678901234567890123456789012345678901234567890123456789012
To install changes, type C-X C-S
```

The first line contains a colon at each tab stop. The remaining lines are present just to help you see where the colons are and know what to do.

Note that the tab stops that control **tab-to-tab-stop** have nothing to do with displaying tab characters in the buffer. See section 2.1 [Characters], page 11, for more information on that.

21.2 Tabs vs. Spaces

Emacs normally uses both tabs and spaces to indent lines. If you prefer, all indentation can be made from spaces only. To request this, set **indent-tabs-mode** to **nil**.

There are also commands to convert tabs to spaces or vice versa, always preserving the columns of all nonblank text. `M-x tabify` scans the region for sequences of spaces, and converts sequences of at least three spaces to tabs if that can be done without changing indentation. `M-x untabify` changes all tabs in the region to appropriate numbers of spaces.

22. Commands for Human Languages

The term *text* has two widespread meanings in our area of the computer field. One is, data that is a sequence of characters. Any file that you edit with Emacs is text, in this sense of the word. The other meaning is more restrictive; it is, a sequence of characters in a human language for humans to read (possibly after processing by a text formatter), as opposed to a program or commands for a program.

Human languages have syntactic/stylistic conventions that can be supported or used to advantage by editor commands: conventions involving words, sentences, paragraphs, and capital letters. This chapter describes Emacs commands for all of these things. In addition, there is a major mode, Text mode, which customizes Emacs in small ways for editing a file of human language text. There are also commands for *filling*, or rearranging paragraphs into lines of approximately equal length.

The commands for moving over and killing words (see section 22.2 [Words], page 101), sentences (see section 22.3 [Sentences], page 103) and paragraphs (see section 22.4 [Paragraphs], page 104) are are primarily intended for human-language text, but are very often useful in editing programs also.

22.1 Text Mode

Editing files of text in a human language ought to be done using Text mode rather than Lisp or Fundamental mode. Invoke `M-x text-mode` to enter Text mode. In Text mode, `TAB` runs the function tab-to-tab-stop, which allows you to use arbitrary tab stops set with `M-x edit-tab-stops` (see section 21.1 [Tab Stops], page 98). Features concerned with comments in programs are turned off except when explicitly invoked. The syntax table is changed so that periods are not considered part of a word, while apostrophes, backspaces and underlines are.

A similar variant mode is Indented Text mode, intended for editing text in which most lines are indented. This mode defines `TAB` to run indent-relative (see chapter 21 [Indentation], page 97), and makes Auto Fill indent the lines it creates. The result is that normally a line made by Auto Filling, or by `LFD`, is indented just like the previous line. Use `M-x indented-text-mode` to select this mode.

22.2 Words

Emacs has commands for moving over or operating on words. By convention, the keys for them are all `Meta-` characters.

`M-f`	Move Forward over a word.
`M-b`	Move Backward over a word.
`M-d`	Kill up to the end of a word.
`M-DEL`	Kill back to the beginning of a word.
`M-@`	Mark the end of the next word.
`M-t`	Transpose two words; drag a word forward or backward across other words.

Notice how these keys form a series that parallels the character-based `C-f`, `C-b`, `C-d`, `C-t` and DEL. `M-@` is related to `C-@`, which is an alias for `C-SPC`.

The commands `Meta-f` (forward-word) and `Meta-b` (backward-word) move forward and backward over words. They are thus analogous to `Control-f` and `Control-b`, which move over single characters. Like their `Control-` analogues, `Meta-f` and `Meta-b` move several words if given an argument. `Meta-f` with a negative argument moves backward, and `Meta-b` with a negative argument moves forward. Forward motion stops right after the last letter of the word, while backward motion stops right before the first letter.

`Meta-d` (kill-word) kills the word after point. To be precise, it kills everything from point to the place `Meta-f` would move to. Thus, if point is in the middle of a word, `Meta-d` kills just the part after point. If some punctuation comes between point and the next word, it is killed along with the word. If you wish to kill only the next word but not the punctuation before it, simply do `Meta-f` to get the end, and kill the word backwards with `Meta-DEL`. `Meta-d` takes arguments just like `Meta-f`.

`Meta-DEL` (backward-kill-word) kills the word before point. It kills everything from point back to where `Meta-b` would move to. If point is after the space in '`FOO, BAR`', then '`FOO, `' is killed. If you wish to kill just '`FOO`', do `Meta-b Meta-d` instead of `Meta-DEL`.

`Meta-t` (transpose-words) exchanges the words before or containing point with the following word. The delimiter characters between the words do not move. For example, '`FOO, BAR`' transposes into '`BAR, FOO`' rather than '`BAR FOO,`'. See section 15.2 [Transposition], page 68, for more on transposition and on arguments to transposition commands.

To operate on the next *n* words with an operation which applies between point and mark, you can either set the mark at point and then move over the words, or you can use the command

Meta-@ (mark-word) which does not move point, but sets the mark where Meta-f would move to. It can be given arguments just like Meta-f.

The word commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be a word delimiter. See section 29.5 [Syntax], page 169.

22.3 Sentences

The Emacs commands for manipulating sentences and paragraphs are mostly on Meta- keys, so as to be like the word-handling commands.

M-a	Move back to the beginning of the sentence.
M-e	Move forward to the end of the sentence.
M-k	Kill forward to the end of the sentence.
C-x DEL	Kill back to the beginning of the sentence.

The commands Meta-a and Meta-e (backward-sentence and forward-sentence) move to the beginning and end of the current sentence, respectively. They were chosen to resemble Control-a and Control-e, which move to the beginning and end of a line. Unlike them, Meta-a and Meta-e if repeated or given numeric arguments move over successive sentences. Emacs considers a sentence to end wherever there is a '.', '?' or '!' followed by the end of a line or two spaces, with any number of ')', ']', ''', or '"' characters allowed in between. A sentence also begins or ends wherever a paragraph begins or ends.

Neither M-a nor M-e moves past the newline or spaces beyond the sentence edge at which it is stopping.

Just as C-a and C-e have a kill command, C-k, to go with them, so M-a and M-e have a corresponding kill command M-k (kill-sentence) which kills from point to the end of the sentence. With minus one as an argument it kills back to the beginning of the sentence. Larger arguments serve as a repeat count.

There is a special command, C-x DEL (backward-kill-sentence) for killing back to the beginning of a sentence, because this is useful when you change your mind in the middle of composing text.

The variable sentence-end controls recognition of the end of a sentence. It is a regexp that matches the last few characters of a sentence, together with the whitespace following the sentence. Its normal value is

```
"[.?!][]\")]*\\($\\|\t\\|  \\)[ \t\n]*"
```

22.4 Paragraphs

The Emacs commands for manipulating paragraphs are also `Meta-` keys.

`M-[`	Move back to previous paragraph beginning.
`M-]`	Move forward to next paragraph end.
`M-h`	Put point and mark around this or next paragraph.

`Meta-[` (backward-paragraph) moves to the beginning of the current or previous paragraph, while `Meta-]` (forward-paragraph) moves to the end of the current or next paragraph. Blank lines and text formatter command lines separate paragraphs and are not part of any paragraph. Also, an indented line starts a new paragraph.

In major modes for programs (as opposed to Text mode), paragraphs begin and end only at blank lines. This makes the paragraph commands continue to be useful even though there are no paragraphs per se.

When there is a fill prefix, then paragraphs are delimited by all lines which don't start with the fill prefix. See section 22.6 [Filling], page 106.

When you wish to operate on a paragraph, you can use the command `Meta-h` (mark-paragraph) to set the region around it. This command puts point at the beginning and mark at the end of the paragraph point was in. If point is between paragraphs (in a run of blank lines, or at a boundary), the paragraph following point is surrounded by point and mark. If there are blank lines preceding the first line of the paragraph, one of these blank lines is included in the region.

Thus, for example, `M-h C-w` kills the paragraph around or after point.

The precise definition of a paragraph boundary is controlled by the variables paragraph-separate and paragraph-start. The value of paragraph-start is a regexp that should match any line that either starts or separates paragraphs. The value of paragraph-separate is another regexp that should match only lines that separate paragraphs without being part of any paragraph. For example, normally **paragraph-start** is `"^[ "t"n"f]"` and **paragraph-separate** is `"^[ "t"f]*$"`.

Normally it is desirable for page boundaries to separate paragraphs. The default values of these variables recognize the usual separator for pages.

22.5 Pages

Files are often thought of as divided into *pages* by the *formfeed* character (^L, octal code 014). For example, if a file is printed on a line printer, each page of the file, in this sense, will start on a new page of paper. Emacs treats a page-separator character just like any other character. It can be inserted with `C-q C-l`, or deleted with `DEL`. Thus, you are free to paginate your file, or not. However, sinc(pages are often meaningful divisions of the file, commands are provided to move over them and operate on them.

`C-x C-p`	Put point and mark around this page (or another page).
`C-x [`	Move point to previous page boundary.
`C-x ]`	Move point to next page boundary.
`C-x l`	Count the lines in this page.

The `C-x [` (**backward-page**) command moves point to immediately after the previous page delimiter. If point is already right after a page delimiter, it skips that one and stops at the previous one. A numeric argument serves as a repeat count. The `C-x ]` (**forward-page**) command moves forward past the next page delimiter.

The `C-x C-p` command (**mark-page**) puts point at the beginning of the current page and the mark at the end. The page delimiter at the end is included (the mark follows it). The page delimiter at the front is excluded (point follows it). This command can be followed by `C-w` to kill a page which is to be moved elsewhere. If it is inserted after a page delimiter, at a place where `C-x ]` or `C-x [` would take you, then the page will be properly delimited before and after once again.

A numeric argument to `C-x C-p` is used to specify which page to go to, relative to the current one. Zero means the current page. One means the next page, and -1 means the previous one.

The `C-x l` command (**count-lines-page**) is good for deciding where to break a page in two. It prints in the echo area the total number of lines in the current page, and then divides it up into those preceding the current line and those following, as in

```
    Page has 96 (72+25) lines
```

Notice that the sum is off by one; this is correct if point is not at the front of a line.

The variable page-delimiter should have as its value a regexp that matches the beginning of a line that separates pages. This is what defines where pages begin. The normal value of this variable is "^“f", which matches a formfeed character at the beginning of a line.

22.6 Filling Text

With Auto Fill mode, text can be *filled* (broken up into lines that fit in a specified width) as you insert it. If you alter existing text it may no longer be properly filled; then explicit commands for filling can be used.

`M-x auto-fill-mode`
: Enable or disable Auto Fill mode.

`SPC`
`RET` In Auto Fill mode, break lines when appropriate.

`M-q` Fill current paragraph.

`M-g` Fill each paragraph in the region.

`M-x fill-region-as-paragraph.`
: Fill the region, considering it as one paragraph.

`M-s` Center a line.

`M-x auto-fill-mode` turns Auto Fill mode on if it was off, or off if it was on. With a positive numeric argument it always turns Auto Fill mode on, and with a negative argument always turns it off. You can see when Auto Fill mode is in effect by the presence of the word '`Fill`' in the mode line, inside the parentheses. Auto Fill mode is a minor mode, turned on or off for each buffer individually. See section 29.1 [Minor Modes], page 159.

In Auto Fill mode, lines are broken automatically at spaces when they get longer than the desired width. Line breaking and rearrangement takes place only when you type SPC or RET. If you wish to insert a space or newline without permitting line-breaking, type `C-q` SPC or `C-q` LFD (recall that a newline is really a linefeed). Also, `C-o` inserts a newline without line breaking.

Auto Fill mode works well with Lisp mode, because when it makes a new line in Lisp mode it indents that line with TAB. If a line ending in a comment gets too long, the text of the comment is split into two comments.

Auto Fill mode does not refill entire paragraphs. It can break lines but cannot merge lines. So editing in the middle of a paragraph can result in a paragraph that is not correctly filled. To refill a paragraph, use the command Meta-q (fill-paragraph). It causes the paragraph that point is inside, or the one after point if point is between paragraphs, to be refilled. All the line-breaks are removed, and then new ones are inserted where necessary. M-q can be undone with C-_. See chapter 12 [Undo], page 51.

To refill many paragraphs, use M-g (fill-region), which divides the region into paragraphs and fills each of them.

Meta-q and Meta-g use the same criteria as Meta-h for finding paragraph boundaries (see section 22.4 [Paragraphs], page 104). For more control, you can use M-x **fill-region-as-paragraph**, which refills everything between point and mark. This command recognizes only blank lines as paragraph separators.

A numeric argument to M-g or M-q causes it to *justify* the text as well as filling it. This means that extra spaces are inserted to make the right margin line up exactly at the fill column. Extra spaces are removed by M-q or M-g with no argument.

The command Meta-s (center-line) centers the current line within the current fill column. With an argument, it centers several lines individually and moves past them.

The maximum line width for filling is in the variable fill-column. This variable has a separate value in each buffer; setting it in one buffer has no effect on any other buffer. The initial value in a new buffer is taken from the variable default-fill-column.

The easiest way to set fill-column is to use the command C-x f (set-fill-column). With no argument, it sets fill-column to the current horizontal position of point. With a numeric argument, it uses that as the new fill column.

To fill a paragraph in which each line starts with a special marker (which might be a few spaces, giving an indented paragraph), use the *fill prefix* feature. The fill prefix is a string which Emacs expects every line to start with, and which is not included in filling. It is stored in the variable fill-prefix.

To specify a fill prefix, move to a line that starts with the desired prefix, put point at the end of the prefix, and give the command C-X . (set-fill-prefix). That's a period after the C-x. To turn off the fill prefix, specify an empty prefix: type C-x . with point at the beginning of a line.

When a fill prefix is in effect, the fill commands remove the fill prefix from each line before filling and insert it on each line after filling. In Auto Fill mode, SPC also inserts the fill prefix on any new line. Lines that do not start with the fill prefix are considered to start paragraphs, both in M-q and the paragraph commands; this is just right if you are using paragraphs with hanging indentation (every line indented except the first one). Lines which are blank or indented once the prefix is removed also separate or start paragraphs; this is what you want if you are writing multi-paragraph comments with a comment delimiter on each line.

Many users like Auto Fill mode and want to use it in all text files. Execute the following Lisp expression, perhaps in your init file, to cause Auto Fill mode to be turned on whenever Text mode is entered:

```
(setq text-mode-hook 'turn-on-auto-fill)
```

22.7 Case Conversion Commands

Emacs has commands for converting either a single word or any arbitrary range of text to upper case or to lower case.

M-l	Convert following word to lower case.
M-u	Convert following word to upper case.
M-c	Capitalize the following word.
C-x C-l	Convert region to lower case.
C-x C-u	Convert region to upper case.

The word conversion commands are the most useful. Meta-l (downcase-word) converts the word after point to lower case, moving past it. Thus, repeating Meta-l converts successive words. Meta-u (upcase-word) converts to all capitals instead, while Meta-c (capitalize-word) puts the first letter of the word into upper case and the rest into lower case. All these commands convert several words at once if given an argument. They are especially convenient for converting a large amount of text from all upper case to mixed case, because you can move through the text using M-l, M-u or M-c on each word as appropriate, occasionally using M-f instead to skip a word.

When given a negative argument, the word case conversion commands apply to the appropriate number of words before point, but do not move point. This is convenient when you have just typed a word in the wrong case. You can give the case conversion command and continue typing.

If a word case conversion command is given in the middle of a word, it applies only to the part of the word which follows point. This is just like what **Meta-d** (kill-word) does. With a negative argument, case conversion applies only to the part of the word before point.

The other basic case conversion commands are **C-x C-u** (upcase-region) and **C-x C-l** (downcase-region), which convert everything between point and mark to the specified case. Point and mark do not move.

23. Editing Programs

Emacs has many commands designed to understand the syntax of programming languages such as Lisp and C. These commands can

- Move over or kill balanced expressions or sexps (see section 23.2 [Lists], page 112).
- Move over or mark top-level balanced expressions (*defuns*, in Lisp; functions, in C).
- Show how parentheses balance (see section 23.5 [Matching], page 120).
- Insert, kill or align comments (see section 23.6 [Comments], page 120).
- Follow the usual indentation conventions of the language (see section 23.4 [Grinding], page 115).

The commands for words, sentences and paragraphs are very useful in editing code even though their canonical application is for editing human language text. Most symbols contain words; sentences can be found in strings and comments. Paragraphs per se are not present in code, but the paragraph commands are useful anyway, because Lisp mode and C mode define paragraphs to begin and end at blank lines. Judicious use of blank lines to make the program clearer will also provide interesting chunks of text for the paragraph commands to work on. See chapter 22 [Text], page 101.

23.1 Major Modes for Programming Languages

Emacs also has major modes for the programming languages Lisp, Scheme (a variant of Lisp), C and Muddle. Ideally, a major mode should be implemented for each programming language that you might want to edit with Emacs; but often the mode for one language can serve for other syntactically similar languages. The language modes that exist are those that someone decided to trouble to write.

There are several forms of Lisp mode, which differ in the way they interface to Lisp execution. See section 24.2 [Lisp Modes], page 132.

Each of the programming language modes defines the `TAB` key to run an indentation function that knows the indentation conventions of that language and updates the current line's indentation accordingly. For example, in C mode `TAB` is bound to `c-indent-line`. `LFD` is normally defined to do `RET` followed by `TAB`; thus, it too indents in a mode-specific fashion.

In most programming languages, indentation is likely to vary from line to line. So the major modes for those languages rebind `DEL` to treat a tab as if it were the equivalent number of spaces (using the command backward-delete-char-untabify). This makes it possible to rub out indentation one column at a time without worrying whether it is made up of spaces or tabs. Use `C-b C-d` to delete a tab character before point, in these modes.

Paragraphs are defined to start only with blank lines so that the paragraph commands can be useful. Auto Fill mode, if enabled in a programming language major mode, indents the new lines which it creates.

23.2 Lists and Sexps

`C-M-f`	Move forward over a sexp.
`C-M-b`	Move Backward over a sexp.
`C-M-k`	Kill sexp forward.
`C-M-u`	Move up and backward in list structure.
`C-M-d`	Move down and forward in list structure.
`C-M-n`	Move forward over a list (parenthetical grouping).
`C-M-P`	Move backward over a list (parenthetical grouping).
`C-M-t`	Transpose expressions.
`C-M-@`	Put mark after following expression.

By convention, Emacs keys for dealing with balanced expressions are usually `Control-Meta-` characters. They tend to be analogous in function to their `Control-` and `Meta-` equivalents. These commands are usually thought of as pertaining to expressions in programming languages, but can be useful with any language in which some sort of parentheses exist (including English).

These commands fall into two classes. Some deal only with *lists* (parenthetical groupings). They see nothing except parentheses, brackets, braces, and escape characters that might be used to quote those. The other commands deal with expressions or *sexps* (short for *s-expression*, the ancient term for a balanced expression in Lisp). A parenthetical grouping is one kind of sexp, but a symbol name is also a sexp, and so is a string. Numbers and character constants can also be sexps. The idea is to define the major mode for a language so that the expressions of that language count as sexps, as much as possible.

Except in Lisp-like languages, not all expressions can be sexps. For example, C mode does not recognize foo + bar as a sexp, even though it *is* a C expression; it recognizes foo as one sexp and bar as another, with the + as punctuation between them. This is a fundamental ambiguity: both foo + bar and foo are legitimate choices for the sexp to move over if point is at the f. Note that (foo + bar) is a sexp in C mode.

Some languages have obscure forms of syntax for expressions that nobody has bothered to make Emacs understand properly.

To move forward over a sexp, use `C-M-f` (forward-sexp). If the first significant character after point is an opening delimiter ('(' in Lisp, '(', '[' or '{' in C), `C-M-f` moves past the matching closing delimiter. If the character begins a symbol, string, or number, `C-M-f` moves over that. If the character after point is a closing delimiter, `C-M-f` just moves past it. (This last is not really moving across an sexp; it is an exception which is included in the definition of `C-M-f` because it is as useful a behavior as anyone can think of for that situation.)

The command `C-M-b` (backward-sexp) moves backward over a sexp. The detailed rules are like those above ıor `C-M-f`, but with directions reversed. If there are any prefix characters (singlequote, backquote and comma, in Lisp) preceding the sexp, `C-M-b` moves back over them as well.

`C-M-f` or `C-M-b` with an argument repeats that operation the specified number of times; with a negative argument, it moves in the opposite direction.

The sexp commands move across comments as if they were whitespace, in languages such as C where the comment-terminator can be recognized. In Lisp, and other languages where comments run until the end of a line, it is very difficult to ignore comments when parsing backwards; therefore, the sexp commands in such languages treat the text of comments as if it were code.

Killing a sexp at a time can be done with `C-M-k` (kill-sexp). `C-M-k` kills the characters that `C-M-f` would move over.

The *list commands* move over lists like the sexp commands but skip blithely over any number of other kinds of sexps (symbols, strings, etc). They are `C-M-n` (forward-list) and `C-M-p` (backward-list). The main reason they are useful is that they usually ignore comments (since the comments usually do not contain any lists).

`C-M-n` and `C-M-p` stay at the same level in parentheses, when that's possible. To move *up* one (or *n*) levels, use `C-M-u` (backward-up-list). `C-M-u` moves backward up past one unmatched opening delimiter. A positive argument serves as a repeat count; a negative argument reverses direction of motion and also requests repetition, so it moves forward and up a level.

To move *down* in list structure, use `C-M-d` (down-list). In Lisp mode, where '(' is the only opening delimiter, this is nearly the same as searching for a '('.

A somewhat random-sounding command which is nevertheless easy to use is `C-M-t` (transpose-sexps), which drags the previous sexp across the next one. An argument serves as a repeat count, and a negative argument drags backwards (thus canceling out the effect of `C-M-t` with a positive argument). An argument of zero, rather than doing nothing, transposes the sexps at the point and the mark.

To make the region be the next sexp in the buffer, use `C-M-@` (mark-sexp) which sets mark at the same place that `C-M-f` would move to. `C-M-@` takes arguments like `C-M-f`. In particular, a negative argument is useful for putting the mark at the beginning of the previous sexp.

The list and sexp commands' understanding of syntax is completely controlled by the syntax table. Any character can, for example, be declared to be an opening delimiter and act like an open parenthesis. See section 29.5 [Syntax], page 169.

23.3 Defuns

In Emacs, a list at the top level in the buffer is called a *defun*. The name derives from the fact that most top level lists in a Lisp file are instances of the special form defun, but any top level list counts as a defun in Emacs parlance regardless of what its contents are, and regardless of the programming language in use. For example, in C, the body of a function definition is a defun.

`C-M-a`	Move to beginning of defun.
`C-M-e`	Move to end of defun.
`C-M-h`	Put region around whole defun.

The commands to move to the beginning and end of the current defun are `C-M-a` (beginning-of-defun) and `C-M-e` (end-of-defun).

If you wish to operate on the current defun, use `C-M-h` (mark-defun) which puts point at the beginning and mark at the end of the current or next defun. For example, this is the easiest way to get ready to move the defun to a different place in the text. In C mode, `C-M-h` runs the function mark-c-function, which is almost the same as mark-defun; the difference is that it backs up over the argument declarations, function name and returned data type so that the entire C function is inside the region.

Emacs assumes that any open-parenthesis found in the leftmost column is the start of a defun. Therefore, **never put an open-parenthesis at the left margin in a Lisp file unless it is the start of a top level list. Never put an open-brace or other opening delimiter at the beginning of a line of C code unless it starts the body of a function.** The most likely problem case is when you want an opening delimiter at the start of a line inside a string. To avoid trouble, put an escape character ('\', in C and Emacs Lisp, '/' in some other Lisp dialects) before the opening delimiter. It will not affect the contents of the string.

In the remotest past, the original Emacs found defuns by moving upward a level of parentheses until there were no more levels to go up. This always required scanning all the way back to the beginning of the buffer, even for a small function. To speed up the operation, Emacs was changed to assume that any '`(`' (or other character assigned the syntactic class of opening-delimiter) at the left margin is the start of a defun. This heuristic is nearly always right and avoids the costly scan. GNU Emacs uses the same convention.

23.4 Indentation for Programs

The best way to keep a program properly indented ("ground") is to use Emacs to re-indent it when it is changed. Emacs has commands to indent properly either a single line, a specified number of lines, or all of the lines inside a single parenthetical grouping.

`TAB`	Adjust indentation of current line.
`LFD`	Equivalent to `RET` followed by `TAB`.
`C-M-q`	Re-indent all the lines within one list.
`C-u TAB`	Shift an entire list rigidly sideways so that its first line is properly indented.
`C-M-\`	Re-indent all lines in the region.

The basic indentation command is `TAB`, which gives the current line the correct indentation as determined from the previous lines. The function that `TAB` runs depends on the major mode; it is lisp-indent-line in Lisp mode, c-indent-line in C mode, etc. These functions understand different syntaxes for different languages, but they all do about the same thing. `TAB` in any programming language major mode inserts or deletes whitespace at the beginning of the current line, independent of where point is in the line. If point is inside the whitespace at the beginning of the line, `TAB` leaves it at the end of that whitespace; otherwise, `TAB` leaves point fixed with respect to the characters around it.

Use `C-q TAB` to insert a tab at point.

When entering a large amount of new code, use LFD (newline-and-indent), which is equivalent to a RET followed by a TAB. LFD creates a blank line, and then gives it the appropriate indentation.

TAB indents the second and following lines of the body of an parenthetical grouping each under the preceding one; therefore, if you alter one line's indentation to be nonstandard, the lines below will tend to follow it. This is the right behavior in cases where the standard result of TAB is unaesthetic.

23.4.1 Indenting Several Lines

When you wish to re-indent code which has been altered or moved to a different level in the list structure, you have several commands available.

You can re-indent the contents of a single list by positioning point before the beginning of it and typing C-M-q (indent-sexp in Lisp mode, indent-c-exp in C mode; also bound to other suitable functions in other modes). The indentation of the line the sexp starts on is not changed; therefore, only the relative indentation within the list, and not its position, is changed. To correct the position as well, type a TAB before the C-M-q.

If the relative indentation within a list is correct but the indentation of its beginning is not, go to the line the list begins on and type C-u TAB. When TAB is given a numeric argument, it moves all the lines in the grouping starting on the current line sideways the same amount that the current line moves. It is clever, though, and does not move lines that start inside strings, or C preprocessor lines when in C mode.

Another way to specify the range to be re-indented is with point and mark. The command C-M-\ (indent-region) applies TAB to every line whose first character is between point and mark.

23.4.2 Customizing Lisp Indentation

The indentation pattern for a Lisp expression can depend on the function called by the expression. For each Lisp function, you can choose among several predefined patterns of indentation, or define an arbitrary one with a Lisp program.

The standard pattern of indentation is as follows: the second line of the expression is indented under the first argument, if that is on the same line as the beginning of the expression; otherwise, the second line is indented underneath the function name. Each following line is indented under the previous line whose nesting depth is the same.

If the variable **lisp-indent-offset** is non-**nil**, it overrides the usual indentation pattern for the second line of an expression, so that such lines are always indented **lisp-indent-offset** more columns than the containing list.

The standard pattern is overridded for certain functions. Functions whose names start with **def** always indent the second line by **lisp-body-indention** extra columns beyond the open-parenthesis starting the expression.

The standard pattern can be overridden in various ways for individual functions, according to the **lisp-indent-hook** property of the function name. There are four possibilities for this property:

nil	This is the same as no property; the standard indentation pattern is used.
defun	The pattern used for function names that start with **def** is used for this function also.
number	The first *number* arguments of the function are *distinguished* arguments; the rest are considered the *body* of the expression. A line in the expression is indented according to whether the first argument on it is distinguished or not. If the argument is part of the body, the line is indented **lisp-body-indent** more columns than the open-parenthesis starting the containing expression. If the argument is distinguished andis either the first or second argument, it is indented *twice* that many extra columns. If the argument is distinguished and not the first or second argument, the standard pattern is followed for that line.
symbol	*symbol* should be a function name; that function is called to calculate the indentation of a line within this expression. The function receives two arguments:

state	The value returned by **parse-partial-sexp** (a Lisp primitive for indentation and nesting computation) when it parses up to the beginning of this line.
pos	The position at which the line being indented begins.

It should return either a number, which is the number of columns of indentation for that line, or a list whose car is such a number. The difference between returning a number and returning a list is that a number says that all following lines at the same nesting level should be indented just like this one; a list says that following lines might call for different indentations. This makes a difference when the indentation is being computed by `C-M-q`; if the value is a number, `C-M-q` need not recalculate indentation for the following lines until the end of the list.

23.4.3 Customizing C Indentation

C does not have anything analogous to particular function names for which special forms of indentation are desirable. However, it has a different need for customization facilities: many different styles of C indentation are in common use. There are six variables you can set to control the style that Emacs C mode will use.

The variable c-indent-level controls the indentation for C statements with respect to the surrounding block. In the example

```
{
  foo ();
```

the difference in indentation between the lines is c-indent-level. Its standard value is 2.

If the open brace beginning the compound statement is not at the beginning of its line, the c-indent-level is added to the indentation of the line, not the column of the open-brace. For example,

```
if (losing) {
  do_this ();
```

One popular indentation style is that which results from setting c-indent-level to 8 and putting open-braces at the end of a line in this way.

c-continued-statement-offset controls the extra indentation for a line that starts within a statement (but not within parentheses or brackets). These lines are usually statements that are within other statements, such as the then-clauses of if statements and the bodies of while statements. This parameter is the difference in indentation between the two lines in

```
if (x == y)
  do_it ();
```

Its standard value is 2. Some popular indentation styles correspond to a value of zero for c-continued-statement-offset.

c-brace-offset is the extra indentation given to a line that starts with an open-brace or close-brace. Its standard value is zero; compare

```
if (x == y)
  {
```

with

```
if (x == y)
  do_it ();
```

if c-brace-offset were set to 4, the first example would become

```
if (x == y)
      {
```

c-argdecl-indent controls the indentation of declarations of the arguments of a C function. It is absolute: argument declarations receive exactly c-argdecl-indent spaces. The standard value is 5, resulting in code like this:

```
char *
index (string, char)
     char *string;
     int char;
```

c-label-offset is the extra indentation given to a line thst contains a label, a case statement, or a default statement. Its standard value is -2, resulting in code like this

```
switch (c)
  {
  case 'x':
```

If c-label-offset were zero, the same code would be indented as

```
switch (c)
  {
    case 'x':
```

This example assumes that the other variables above also have their standard values.

I strongly recommend that you try out the indentation style produced by the standard settings of these variables, together with putting open braces on separate lines. You can see how it looks in all the C source files of GNU Emacs.

One other variable, c-auto-newline, does not affect the style of indentation that is used, but makes insertion of certain characters insert newlines automatically. When this variable is non-nil, newlines are inserted both before and after braces that you insert, and after colons and semicolons. Correct C indentation is done on all the lines that are made this way.

23.5 Automatic Display Of Matching Parentheses

The Emacs parenthesis-matching feature is designed to show automatically how parentheses match in the text. When ever a self-inserting character that is a closing delimiter is typed, the cursor moves momentarily to the location of the matching opening delimiter, provided that is on the screen. If it is not on the screen, some text starting with that opening delimiter is displayed in the echo area. Either way, you can tell what grouping is being closed off.

In Lisp, automatic matching applies only to parentheses. In C, it applies to braces and brackets too. Emacs knows which characters to regard as matching delimiters based on the syntax table, which is set by the major mode. See section 29.5 [Syntax], page 169.

If the opening delimiter and closing delimiter are mismatched—such as, in '[x)'—a warning message is displayed in the echo area. The correct matches are specified in the syntax table.

Two variables control parenthesis match display. blink-matching-paren turns the feature on or off; nil turns it off, but the default is t to turn match display on. blink-matching-paren-distance specifies how many characters back to search to find the matching opening delimiter. If the match is not found in that far, scanning stops, and nothing is displayed. This is to prevent scanning for the matching delimiter from wasting lots of time when there is none.

23.6 Manipulating Comments

The comment commands insert, kill and align comments. There are also commands for moving through existing code and inserting comments.

M-;	Insert or align comment.

C-x ;	Set comment column.
C-u - C-x ;	Kill comment on current line. With region, kill comments in region.
M-LFD	Like RET followed by inserting and aligning a comment-start string.

The command that creates a comment is Meta-; (indent-for-comment). If there is no comment already on the line, a new comment is created, aligned at a specific column called the *comment column*. The comment is created by inserting whatever string Emacs thinks should start comments in the current major mode. Point is left after the comment-starting string. If the text of the line goes past the comment column, then the indentation is done to a suitable boundary (usually, at least one space is inserted). If the major mode has specified a string to terminate comments, that is inserted after point, to keep the syntax valid.

Meta-; can also be used to align an existing comment. If a line already contains the string that starts comments, then M-; just moves point after it and re-indents it to the right column. Exception: comments starting in column 0 are not moved. Also, in particular modes, there are special rules for indenting certain kinds of comments in certain contexts.

Even when an existing comment is properly aligned, M-; is still useful for moving directly to the start of the comment.

C-u - C-x ; (kill-comment) kills the comment on the current line, if there is one. The indentation before the start of the comment is killed as well. If there does not appear to be a comment in the line, nothing is done. To reinsert the comment on another line, move to the end of that line, do C-y, and then do M-; to realign it. Note that C-u - C-x ; is not a distinct key; it is C-x ; (set-comment-column) with a negative argument. That command is programmed so that when it receives a negative argument it calls kill-comment. However, kill-comment is a valid command which you could bind directly to a key if you wanted to.

23.6.1 Multiple Lines of Comments

If you wish to align a large number of comments, give Meta-; an argument, and it indents what comments exist on that many lines, creating none. Point is left after the last line processed (unlike the no-argument case).

If you are typing a comment and find that you wish to continue it on another line, you can use the command Meta-LFD (indent-new-comment-line), which terminates the comment you are typing, creates a new blank line afterward, and begins a new comment indented under the old one. When

Auto Fill mode is on, going past the fill column while typing a comment causes the comment to be continued in just this fashion. If point is not at the end of the line when M-LFD is typed, the text on the rest of the line becomes part of the new comment line.

23.6.2 Double and Triple Semicolons in Lisp

In Lisp code there are conventions for comments which start with more than one semicolon. Comments which start with two semicolons are indented as if they were lines of code, instead of at the comment column. Comments which start with three semicolons are supposed to start at the left margin. Emacs understands these conventions by indenting a double-semicolon comment using TAB, and by not changing the indentation of a triple-semicolon comment at all. (Actually, this rule applies whenever the comment starter is a single character and is duplicated).

23.6.3 Options Controlling Comments

The comment column is stored in the variable comment-column. You can set it to a number explicitly; note that the value is in pixels, not characters. Alternatively, the command C-x ; (set-comment-column) sets the comment column to the column point is at. C-u C-x ; sets the comment column to match the last comment before point in the buffer, and then does a Meta-; to align the current line's comment under the previous one. Note that C-u - C-x ; runs the function kill-comment as described above.

Many major modes supply default local values for the comment column. Its value is local to each buffer, so changing it in one buffer does not other buffers. See section 29.2 [Variables], page 159.

The comment commands recognize comments based on the regular expression that is the value of the variable comment-start-skip. This regexp should not match the null string. It may match more than the comment starting delimiter in the strictest sense of the word; for example, in C mode the value of the variable is "/*+ *", which matches extra stars and spaces after the '/*' itself.

When a comment command makes a new comment, it inserts the value of comment-start to begin it. The value of comment-end is inserted after point, so that it will follow the text that you will insert into the comment.

comment-multi-line controls how M-LFD indent-new-comment-line behaves when used inside a comment. If comment-multi-line is nil, as it normally is, then the comment on the starting line is terminated and a new comment is started on the new following line. If comment-multi-line is not nil, then the new following line is set up as part of the same comment that was found on the starting line. This is done by not inserting a terminator there, and not inserting a starter on the new line.

The variable comment-indent-hook should contain a function that will be called to compute the indentation for a newly inserted comment or for aligning an existing comment. It is set differently by various major modes. The function is called with no arguments, but with point at the beginning of the comment, or at the end of a line if a new comment is to be inserted. It should return the column in which the comment ought to start. In Lisp mode, the indent hook function can base its decision on how many semicolons begin an existing comment, and on the code in the preceding lines.

23.7 Editing Without Unbalanced Parentheses

`M-(`	Put parentheses around next sexp(s).
`M-)`	Move past next close parenthesis and re-indent.

The commands `M-(` (insert-parentheses) and `M-)` (move-over-close-and-reindent) are designed to facilitate a style of editing which keeps parentheses balanced at all times. `M-(` inserts a pair of parentheses, either together as in '()', or, if given an argument, around the next several sexps, and leaves point after the open parenthesis. Instead of typing `( F O O )`, you can type `M-( F O O`, which has the same effect except for leaving the cursor before the close parenthesis. Then you would type `M-)`, which moves past the close parenthesis, deleting any indentation preceding it (in this example there is none), and indenting with LFD after it.

23.8 Documentation Commands

As you edit Lisp code to be run in Emacs, the commands `C-h f` (describe-function) and `C-h v` (describe-variable) can be used to print documentation of functions and variables that you want to call. These commands use the minibuffer to read the name of a function or variable to document, and display the documentation in a window.

For extra convenience, these commands provide default arguments based on the code in the neighborhood of point. `C-h f` sets the default to the function called in the innermost list containing point. `C-h v` uses the symbol name around or adjacent to point as its default.

Documentation on Unix commands, system calls and libraries can be obtained with the `M-x manual-entry` command. This reads a topic as an argument, and displays the text on that topic from the Unix manual. **manual-entry** always searches all 8 sections of the manual, and concatenates all the entries that are found. For example, the topic '`termcap`' finds the description of the termcap library from section 3, followed by the description of the termcap data base from section 5.

23.9 Change Logs

The Emacs command `M-x add-change-log-entry` helps you keep a record of when and why you have changed a program. It assumes that you have a file in which you write a chronological sequence of entries describing individual changes. The default is to store the change entries in a file called **ChangeLog** in the same directory as the file you are editing. The same **ChangeLog** file therefore records changes in all the files in the directory.

A change log entry starts with a header line that contains a star followed by your name and the current date. Aside from these header lines, every line in the change log starts with a tab. One entry can describe several changes; each change starts with a line starting with a tab and a star. `M-x add-change-log-entry` visits the change log file and creates a new entry unless the most recent entry is for today's date and your name. In either case, it adds a new line to start the description of another change just after the header line of the entry. When `M-x add-change-log-entry` is finished, all is prepared for you to edit in the description of what you changed and how. You must then save the change log file yourself.

The change log file is always visited in Indented Text mode, which means that LFD and auto-filling indent each new line like the previous line. This is convenient for entering the contents of an entry, which must all be indented.

Here is an example of the formatting conventions used in the change log for Emacs:

```
Wed Jun 26 19:29:32 1985  Richard M. Stallman  (rms at mit-prep)

        * xdisp.c (try_window_id):
        If C-k is done at end of next-to-last line,
        this fn updates window_end_vpos and cannot leave
        window_end_pos nonnegative (it is zero, in fact).
        If display is preempted before lines are output,
        this is inconsistent.  Fix by setting
        blank_end_of_window to nonzero.

Tue Jun 25 05:25:33 1985  Richard M. Stallman  (rms at mit-prep)
```

```
* cmds.c (Fnewline):
Call the auto fill hook if appropriate.

* xdisp.c (try_window_id):
If dot is found by compute_motion after xp, record that
permanently.  If display_text_line sets dot position wrong
(case where like is killed, dot is at eob and that line is
not displayed), detect and set it again in final compute_motion.
```

23.10 Tag Tables

A *tag table* is a description of how a multi-file program is broken up into files. It lists the names of the component files and the names and positions of the functions in each file. Grouping the related fil‹s makes it possible to search or replace through all the files with one command. Recording the function names and positions makes possible the Meta-. command which you can use to find the definition of a function without having to know which of the files it is in.

Tag tables are stored in files called *tag table files*. The conventional name for a tag table file is TAGS.

Just what names from the described files are recorded in the tag table depends on the programming language of the described file. They normally include all functions and subroutines, and may also include global variables, data types, and anything else convenient. In any case, each name recorded is called a *tag*.

In Lisp code, any function defined with defun, any variable defined with defvar or defconst, and in general the first argument of any expression that starts with (def in column zero, is a tag.

In C code, any C function is a tag, and so is any typedef if -t is specified when the tag table is constructed.

In Fortran code, functions and subroutines are tags.

23.10.1 Creating Tag Tables

The etags program is used to create a tag table file. It knows the syntax of C, Fortran and Emacs Lisp. To use etags, type

```
etags inputfiles...
```

as a shell command. It reads the specified files and writes a tag table named TAGS in the current working directory. **etags** recognizes the language used in an input file based on its file name and contents; there are no switches for specifying the language. The -t switch tells etags to record typedefs in C code as tags.

Each entry in the tag table records the name of one tag, the position in its file of that tag's definition, and (implicitly) the name of the file that tag is defined in.

If the tag table data become outdated, due to changes in the files described in the table, the way to update the tag table is the same way it was made in the first place. It is not necessary to do this often.

If the tag table fails to record a tag, or records it for the wrong file, then Emacs cannot possibly find its definition. However, if the position recorded in the tag table becomes a little bit wrong (due to some editing in the file that the tag definition is in), the only consequence is to slow down finding the tag slightly. Even if the stored position is very wrong, Emacs will still find the tag, but it must search the entire file for it.

So you should update a tag table when you define new tags that you want to have listed, or when you move tag definitions from one file to another, or when changes become substantial. Normally there is no need to update the tag table after each edit, or even every day.

23.10.2 Selecting a Tag Table

Emacs has at any time one *selected* tag table, and all the commands for working with tag tables use the selected one. To select a tag table, type `M-x visit-tag-table`, which reads the tag table file name as an argument. The name TAGS in the default directory is used as the default file name.

All this command does is store the file name in the variable tags-file-name. Emacs does not actually read in the tag table contents until you try to use them. Setting this variable yourself is just as good as using visit-tag-table. The variable's initial value is nil; this value tells all the commands for working with tag tables that they must ask for a tag table file name to use.

23.10.3 Finding a Tag

The most important thing that a tag table enables you to do is to find the definition of a specific tag.

`M-. tag` Find first definition of *tag*.

`C-u M-.` Find next alternate definition of last tag specified.

`C-x 4 . tag`
: Find first definition of *tag*, but display it in another window.

`M-.` (find-tag) is the command to find the definition of a specified tag. It searches through the tag table for that tag, as a string, and then uses the tag table info to determine the file that the definition is in and the approximate character position in the file of the definition. Then find-tag visits that file, moves point to the approximate character position, and stars searching ever-increasing distances away for the the text that should appear at the beginning of the definition.

If an empty argument is given (just type RET), the sexp in the buffer before or around point is used as the name of the tag to find. See section 23.2 [Lists], page 112, for info on sexps.

The argument to find-tag need not be the whole tag name; it can be a substring of a tag name. However, there can be many tag names containing the substring you specify. Since find-tag works by searching the text of the tag table, it finds the first tag in the table that the specified substring appears in. The way to find other tags that match the substring is to give find-tag a numeric argument, as in `M-0 M-.`; this does not read a tag name, but continues searching the tag table's text for another tag containing the same substring last used.

Like most commands that can switch buffers, find-tag has another similar command that displays the new buffer in another window. `C-x 4 .` invokes the function find-tag-other-window.

23.10.4 Searching and Replacing with Tag Tables

The commands in this section visit and search all the files listed in the selected tag table, one by one.

`M-x tags-search`
: Search for the specified regexp through the files in the selected tag table.

`M-x tags-query-replace`
: Perform a query-replace on each file in the selected tag table.

`M-,` Restart one of the commands above, from the current location of point (`M-COMMA`, tags-loop-continue).

`M-x tags-search` reads a regexp using the minibuffer, then visits the files of the selected tag table one by one, and searches through each one for that regexp. As soon as an occurrence is found, tags-search returns.

Having found one match, you probably want to find all the rest. To find one more match, type `M-,` (tags-loop-continue) to resume the tags-search. This searches the rest of the current buffer, followed by the remaining files of the tag table.

`M-x tags-query-replace` performs a single query-replace through all the files in the tag table. It reads a string to search for and a string to replace with, just like ordinary `M-x query-replace`. It searches much like `M-x tags-search` but repeatedly, processing matches according to your input. See section 14.7 [Replace], page 63, for more information on query-replace.

It is possible to get through all the files in the tag table with a single invocation of `M-x tags-query-replace`. But since any unrecognized character causes the command to exit, you may need to continue where you left off. `M-,` can be used for this. `M-,` resumes the last tags search or replace command that you did.

23.10.5 Stepping Through a Tag Table

If you wish to process all the files in the selected tag table, but `M-x tags search` and `M-x tags-query-replace` in particular are not what you want, you can use `M-x next-file`.

`M-x next-file`
: Visit the next file in the selected tag table.

`C-u M-x next-file`
: With a numeric argument, regardless of its value, visit the first file in the tag table.

23.10.6 Tag Table Inquiries

`M-x list-tags`
: Display a list of the tags defined in a specific program file.

`M-x tags-apropos`
: Display a list of all tags matching a specified regexp.

`M-x list-tags` reads the name of one of the files described by the selected tag table, and displays a list of all the tags defined in that file. The "file name" argument is really just a string to compare against the names recorded in the tag table; it is read as a string rather than as a file name. Therefore, completion and defaulting are not available, and you must enter the string the same way it appears in the tag table. Do not include a directory as part of the file name unless the file name recorded in the tag table includes a directory.

`M-x tags-apropos` is like apropos for tags. It reads a regexp, then finds all the tags in the selected tag table whose entries match that regexp, and displays the tag names found.

24. Compiling, Running and Testing Programs

The previous chapter discusses the Emacs commands that are useful for making changes in programs. This chapter deals with commands that assist in the larger process of developing and maintaining programs.

24.1 Running 'make' or Other Compilers

Emacs can run compilers for noninteractive languages such as C and Fortran as inferior processes, feeding the error log into an Emacs buffer. It can also parse the error messages and visit the files in which errors are found, moving point right to the line where the error occurred.

To run make or other compiler, do `M-x compile`. This command reads a shell command line using the minibuffer, and then executes the specified command line in an inferior shell with output going to the buffer named *compilation*. The current buffer's default directory is used as the working directory for the execution of the command; normally, therefore, the makefile comes from this directory.

When the shell command line is read, the minibuffer appears containing a default command line, which is the command you used the last time you did `M-x compile`. If you type just RET, the same command line is used again. The first `M-x compile` provides make -k as the default. The default is taken from the variable compile-command; if the appropriate compilation command for a file is something other than make -k, it can be useful to have the file specify a local value for compile-command (see section 29.2 [Variables], page 159).

Starting a compilation causes the buffer *compilation* to be displayed in another window but not selected. The mode line tells you whether compilation is finished, with the word 'run' or 'exit' inside the parentheses. You do not have to keep this buffer visible; compilation continues in any case.

To kill the compilation process, do `M-x kill-compilation`. You will see that the mode line of the *compilation* buffer changes to say 'signal' instead of 'run'. Starting a new compilation also kills any running compilation, as only one can exist at any time. However, this requires confirmation before actually killing a compilation that is running.

To parse the compiler error messages, type `` C-x ` `` (next-error). It displays the buffer *compilation* in one window and the buffer in which the next error occurred in another window. Point in that buffer is moved to the line where the error was found. The corresponding error message is scrolled to the top of the window in which ***compilation*** is displayed.

The first time `C-x '` is used, after the start of a compilation, it parses all the error messages, visits all the files that have error messages, and makes markers pointing at the lines that the error messages refer to. Then it moves to the first error message location. Subsequent uses of `C-x '` advance down the data set up by the first use. When the preparsed error messages are exhausted, the next `C-x '` checks for any more error messages that have come in; this is useful if you start editing the compiler errors while the compilation is still going on. If no more error messages have come in, `C-x '` gets a Lisp error.

`C-u C-x '` discards the preparsed error message data and parses the ***compilation*** buffer over again, then displaying the first error. This way, you can process the same set of errors again.

24.2 Major Modes for Lisp

Emacs has four different major modes for Lisp. They are the same in terms of editing commands, but differ in the commands for executing Lisp expressions.

Emacs-Lisp mode
: The mode for editing source files of programs to run in Emacs Lisp. This mode defines `C-M-x` to evaluate the current defun. See section 24.3 [Lisp Libraries], page 132.

Lisp Interaction mode
: The mode for an interactive session with Emacs Lisp. It defines `LFD` to evaluate the sexp before point and insert its value in the buffer. See section 24.6 [Lisp Interaction], page 137.

Lisp mode
: The mode for editing source files of programs that run in Lisps other than Emacs Lisp. This mode defines `C-M-x` to send the current defun to an inferior Lisp process. See section 24.7 [External Lisp], page 137.

Inferior Lisp mode
: The mode for an interactive session with an inferior Lisp process.

24.3 Libraries of Lisp Code for Emacs

Lisp code for Emacs editing commands is stored in files whose names conventionally end in .el.

This ending tells Emacs to edit them in Emacs-Lisp mode, so you can use the `C-M-x` command described in the following section to install changed functions.

Only the maintainers of such a file will want to edit its contents or evaluating text from it, but every user must be able to load the file. This is done with `M-x load`.

`M-x load` reads a file name using the minibuffer and executes the specified file as Lisp code. But it has an important difference from all other Emacs commands that read file names: it searches a sequence of directories, and tries three file names in each directory.

The argument you give to `M-x load` is usually not the full file name. Usually you omit the .el that the file name ends in. `M-x load` tries three file names in each directory: first, the name you specified; second, that name with .elc appended; third, that name with .el appended. A .elc file would be the result of compiling the Lisp file into byte code; it is loaded if possible in preference to the Lisp file itself because the compiled file will load and run faster.

The sequence of directories searched by `M-x load` is specified by the variable load-path, a list of strings that are directory names. Normally the first element of this list is nil, which means to search the current default directory at that point; the remaining elements are the names of the directories in which the Lisp code of Emacs itself is stored. Therefore, you can load an installed Emacs library without having to specify a directory name.

Often you do not have to run the load command yourself, because the commands in a library have permanent definitions to *autoload* that library. Running any of those commands causes load to be called to load the library; this replaces the autoload definitions with the real ones from the library.

The way to make a byte-code compiled file from an Emacs-Lisp source file is with `M-x byte-compile-file`. The default argument for this function is the file visited in the current buffer. It reads the specified file, compiles it into byte code, and writes an output file whose name is made by appending `c` to the input file name. Thus, the file rmail.el would be compiled into rmail.elc.

To recompile the changed Lisp files in a directory, use `M-x byte-recompile-directory`. Specify just the directory name as an argument. Each .el file that has been byte-compiled before is byte-compiled again if it has changed since the previous compilation. A numeric argument to this command tells it to offer to compile each .el file that has not already been compiled. You must answer `Y` or `N` to each offer.

GNU Emacs can run Mocklisp files by converting them to Emacs Lisp first. To convert a

Mocklisp file, visit it and then type `M-x convert-mocklisp-buffer`. Then save the resulting buffer of Lisp code in a file whose name ends in .el and use the new file as a Lisp library.

24.4 Evaluating Emacs-Lisp Expressions

Lisp programs intended to be run in Emacs should be edited in Emacs-Lisp mode; normally this will happen based on the file name that ends in .el. By contrast, Lisp mode itself is used for editing Lisp programs intended for other Lisp systems. Emacs-Lisp mode can be selected with the command `M-x emacs-lisp-mode`.

For testing of Lisp programs to run in Emacs, it is useful to be able to evaluate part of the program as it is found in the Emacs buffer. For example, after changing the text of a Lisp function definition, evaluating the definition installs the change for future calls to the function. Evaluation of Lisp expressions is also useful in any kind of editing task for invoking noninteractive functions (functions that are not commands).

`M-ESC` Read a Lisp expression in the minibuffer, evaluate it, and print the value in the minibuffer.

`C-x C-e` Evaluate the Lisp expression before point, and print the value in the minibuffer.

`C-M-x` Evaluate the defun containing or after point, and print the value in the minibuffer.

`M-x eval-region`
Evaluate all the Lisp expressions in the region.

`M-x eval-current-buffer`
Evaluate all the Lisp expressions in the buffer.

`M-ESC` (eval-expression) is the most basic command for evaluating a Lisp expression interactively. It reads the expression using the minibuffer, so you can execute any expression on a buffer regardless of what the buffer contains. When the expression is evaluated, the current buffer is once again the buffer that was current when `M-ESC` was typed.

`M-ESC` can easily confuse users who do not understand it, especially on keyboards with autorepeat where it can result from holding down the ESC key for too long. Therefore, eval-expression is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; once you enable the command, confirmation will no longer be required for it. See section 29.4.4 [Disabling], page 169.

In Emacs-Lisp mode, the key `C-M-x` is bound to the function eval-defun, which parses the defun containing or following point as a Lisp expression and evaluates it. The value is printed in the echo area. This command is convenient for installing in the Lisp environment changes that you have just made in the text of a function definition.

The command `C-x C-e` (eval-last-sexp) performs a similar job but is available in all major modes, not just Emacs-Lisp mode. It finds the sexp before point, reads it as a Lisp expression, evaluates it, and prints the value in the echo area. It is sometimes useful to type in an expression and then, with point still after it, type `C-x C-e`.

If eval-defun or `C-x C-e` is given a numeric argument, it prints the value by insertion into the current buffer at point, rather than in the echo area. The argument value does not matter.

The most general command for evaluating Lisp expressions from a buffer is eval-region. `M-x eval-region` parses the text of the region as one or more Lisp expressions, evaluating them one by one. `M-x eval-current-buffer` is similar but evaluates the entire buffer. This is a reasonable way to install the contents of a file of Lisp code that you are just ready to test. After finding and fixing a bug, use `C-M-x` on each function that you change, to keep the Lisp world in step with the source file.

24.5 The Lisp Debugger

GNU Emacs contains a debugger for Lisp programs executing inside it. This debugger is normally not used; many commands frequently get Lisp errors when invoked in inappropriate contexts (such as `C-f` at the end of the buffer) and it would be very unpleasant for that to enter a special debugging mode. When you want to make Lisp errors invoke the debugger, you must set the variable debug-on-error to non-nil. Quitting with `C-g` is not considered an error, and debug-on-error has no effect on the handling of `C-g`. However, if you set debug-on-quit non-nil, `C-g` will invoke the debugger. This can be useful for debugging an infinite loop; type `C-g` once the loop has had time to reach its steady state. debug-on-quit has no effect on errors.

You can also cause the debugger to be entered when a specified function is called, or at a particular place in Lisp code. Use `M-x debug-on-entry` with argument *fun-name* to cause function *fun-name* to enter the debugger as soon as it is called. Use `M-x cancel-debug-on-entry` to make the function stop entering the debugger when called. (Redefining the function also does this.) To enter the debugger from some other place in Lisp code, you must insert the expression debug there and install the changed code with `C-M-x`. See section 24.4 [Lisp Eval], page 134.

When the debugger is entered, it displays the previously selected buffer in one window and a backtrace buffer in another window. The backtrace buffer contains one line for each level of Lisp function execution currently going on. At the beginning of this buffer is a message describing the reason that the debugger was invoked (such as, what error message if it was invoked due to an error).

The backtrace buffer is read-only, and is in a special major mode, Backtrace mode, in which letters are defined as debugger commands. The usual Emacs editing commands are available; you can switch windows to examine the buffer that was being edited at the time of the error, and you can also switch buffers. visit files, and do any other sort of editing. However, the debugger is a recursive editing level (see section 27.1 [Recursive Edit], page 149) and it is wise to go back to the backtrace buffer and exit the debugger officially when you don't want to use it any more.

The contents of the backtrace buffer show you the functions that are executing and the arguments that were given to them. It has the additional purpose of allowing yo to specify a stack frame by moving point to the line describing that frame. The frame whose line point is on is considered the *current frame*. Some of the debugger commands operate on the current frame. Debugger commands are mainly used for stepping through code an expression at a time. Here is a list of them.

`b`
: Set up to enter the debugger when the current frame is exited. Frames that will invoke the debugger on exit are marked with stars.

`c`
: Exit the debugger and continue execution. If the debugger was entered due to an error or quit, the usual error or quit handling takes place after you exit the debugger. Otherwise, execution continues.

`d`
: Continue execution, but enter the debugger the next time a Lisp function is called. This allows you to step through the subexpressions of an expression, seeing what values the subexpressions compute and what else they do

  The stack frame made for the function call which enters the debugger in this way will be marked automatically for the debugger to be called when the frame is exited. You can use the `u` command to cancel this mark.

`e`
: Read a Lisp expression in the minibuffer, evaluate it, and print the value in the echo area. The same as the command `M-ESC`, except that `e` is not normally disabled like `M-ESC`.

`u`
: Don't enter the debugger when the current frame is exited. This cancels a `b` command on that frame.

`q`
: Terminate the program being debugged; return to top level Emacs command execution.

`r`
: Return a value from the debugger. The value is computed by reading an expression with the minibuffer and evaluating it. The value returned by the debugger makes a difference when the debugger was invoked due to exit from a Lisp call frame (as

requested with b); then the value specified in the r command is used as the value of that frame.

24.6 Lisp Interaction Buffers

The buffer *scratch* which is selected when Emacs starts up is provided for evaluating Lisp expressions interactively inside Emacs. Both the expressions you evaluate and their output goes in the buffer.

The *scratch* buffer's major mode is Lisp Interaction mode, which is the same as Emacs-Lisp mode except for one command, LFD. In Emacs-Lisp mode, LFD is an indentation command, as usual. In Lisp Interaction mode, LFD is bound to eval-print-last-sexp. This function reads the Lisp expression before point, evaluates it, and inserts the value in printed representation before point.

Thus, the way to use the *scratch* buffer is to insert Lisp expressions at the end, ending each one with LFD so that it will be evaluated. The result is a complete typescript of the expressions you have evaluated and their values.

The rationale for this feature is that Emacs must have a buffer when it starts up, but that buffer is not useful for editing files since a new buffer is made for every file that you visit. The Lisp interpreter typescript is the most useful thing I can think of for the initial buffer to do. M-x lisp-interaction-mode will put any buffer in Lisp Interaction mode.

24.7 Running an External Lisp

Emacs has facilities for running programs in other Lisp systems. You can run a Lisp process as an inferior of Emacs, and pass expressions to it to be evaluated. You can also pass changed function definitions directly from the Emacs buffers in which you edit the Lisp programs to the inferior Lisp process.

To run an inferior Lisp process, type M-x run-lisp. This runs the program named lisp, the same program you would run by typing lisp as a shell command, with both input and output going through an Emacs buffer named *lisp*. That is to say, any "terminal output" from Lisp will go into the buffer, advancing point, and any "terminal input" for Lisp comes from text in the buffer. To give input to Lisp, go to the end of the buffer and type the input, terminated by RET. The *lisp* buffer is in Inferior Lisp mode, a mode which has all the special characteristics of Lisp mode and Shell mode (see section 28.1 [Shell], page 153).

For the source files of programs to run in external Lisps, use Lisp mode. This mode can be selected with `M-x lisp-mode`, and is used automatically for files whose names end in .l or .lisp, as most Lisp systems usually expect.

When you edit a function in a Lisp program you are running, the easiest way to send the changed definition to the inferior Lisp process is the key `C-M-x`. In Lisp mode, this runs the function **lisp-send-defun**, which finds the defun around or following point and sends it as input to the Lisp process. (Emacs can send input to any inferior process regardless of what buffer is current.)

Contrast the meanings of `C-M-x` in Lisp mode (for editing programs to be run in another Lisp system) and Emacs-Lisp mode (for editing Lisp programs to be run in Emacs): in both modes it has the effect of installing the function definition that point is in, but the way of doing so is different according to where the relevant Lisp environment is found. See section 24.2 [Lisp Modes], page 132.

25. Abbrevs

An *abbrev* is a word which changes (*expands*), if you insert it, into some different text. Abbrevs are defined by the user to expand in specific ways. For example, you might define '`foo`' as an abbrev expanding to '`find outer otter`'. With this abbrev defined, you would be able to get '`find outer otter `' into the buffer by typing `f o o SPC`.

Abbrevs expand only when Abbrev mode (a minor mode) is enabled. Disabling Abbrev mode does not cause abbrev definitions to be forgotten, but they do not expand until Abbrev mode is enabled again. The command `M-x abbrev-mode` toggles Abbrev mode; with a numeric argument, it turns Abbrev mode on if the argument is positive, off otherwise. See section 29.1 [Minor Modes], page 159. abbrev-mode is also a variable; Abbrev mode is on when the variable is non-nil.

Abbrev d^finitions can be *mode-specific*—active only in one major mode. Abbrevs can also have *global* definitions that are active in all major modes. The same abbrev can have a global definition and various mode-specific definitions for different major modes. A mode specific definition for the current major mode overrides a global definition.

Abbrevs can be defined interactively during the editing session. Lists of abbrev definitions can also be saved in files and reloaded in later sessions. Some users keep extensive lists of abbrevs that they load in every session.

25.1 Defining Abbrevs

`C-x +` Define an abbrev to expand into some text before point.

`C-x C-a` Similar, but define an abbrev available only in the current major mode.

`M-x inverse-add-global-abbrev`
: Define a word in the buffer as an abbrev.

`M-x inverse-add-mode-abbrev`
: Define a word in the buffer as a mode-specific abbrev.

`M-x kill-all-abbrevs`
: After this command, there are no abbrev definitions in effect.

The usual way to define an abbrev is to enter the text you want the abbrev to expand to, position point after it, and type `C-x +` (add-global-abbrev). This reads the abbrev itself using the minibuffer, and then defines it as an abbrev for one or more words before point. Use a numeric argument to say how many words before point should be taken as the expansion. For example, to define the abbrev 'foo' as mentioned above, insert the text 'find outer otter' and then type C-u 3 C-x + f o o RET.

An argument of zero to C-x + means to use the contents of the region as the expansion of the abbrev being defined.

The command C-x C-a (add-mode-abbrev) is similar, but defines a mode-specific abbrev. Mode specific abbrevs are active only in a particular major mode. C-x C-a defines an abbrev for the major mode in effect at the time C-x C-a is typed. The arguments work the same as for C-x +.

If the text of the abbrev you want is already in the buffer instead of the expansion, use command C-x - (inverse-add-global-abbrev) instead of C-x +, or use C-x C-h (inverse-add-mode-abbrev) instead of C-x C-a. These commands are called "inverse" because they invert the meaning of the argument found in the buffer and the argument read using the minibuffer.

To change the definition of an abbrev, just add the new definition. You will be asked to confirm if the abbrev has a prior definition. To remove an abbrev definition, give a negative argument to C-x + or C-x C-a. You must choose the command to specify whether to kill a global definition or a mode-specific definition for the current mode, since those two definitions are independent for one abbrev.

M-x kill-all-abbrevs removes all the abbrev definitions there are.

25.2 Controlling Abbrev Expansion

An abbrev expands whenever it is present in the buffer just before point and a self-inserting punctuation character (SPC, comma, etc.) is typed. Most often the way an abbrev is used is to insert the abbrev followed by punctuation.

Abbrev expansion preserves case; thus, 'foo' expands into 'find outer otter'; 'Foo' into 'Find outer otter', and 'FOO' into 'FIND OUTER OTTER' or 'Find Outer Otter' according to the variable abbrev-all-caps (a non-nil value chooses the first of the two expansions).

These two commands are used to control abbrev expansion:

M-' Separate a prefix from a following abbrev to be expanded.

M-x unexpand-abbrev

Undo last abbrev expansion.

`M-x expand-region-abbrevs`
: Expand some or all abbrevs found in the region.

You may wish to expand an abbrev with a prefix attached; for example, if 'cnst' expands into 'construction', you might want to use it to enter 'reconstruction'. It does not work to type recnst, because that is not necessarily a defined abbrev. What does work is to use the command M-' (abbrev-prefix-mark) in between the prefix 're' and the abbrev 'cnst'. First, insert 're'. Then type M-'; this inserts a minus sign in the buffer to indicate that it has done its work. Then insert the abbrev 'cnst'; the buffer now contains 're-cnst'. Now insert a punctuation character to expand the abbrev 'cnst' into 'construction'. The minus sign is deleted at this point, because M-' left word for this to be done. The resulting text is the desired 'reconstruction'.

If you actually want the text of the abbrev in the buffer, rather than its expansion, you can accomplish this by inserting the following punctuation with C-q. Thus, foo C-q - leaves 'foo-' in the buffer.

If you expand an abbrev by mistake, you can undo the expansion (replace the expansion by the original abbrev text) with M-x unexpand-abbrev. C-_ (undo) can also be used to undo the expansion; but first it will undo the insertion of the following punctuation character!

M-x expand-region-abbrevs searches through the region for defined abbrevs, and for each one found offers to replace it with its expansion. This command is useful if you have typed in text using abbrevs but forgot to turn on Abbrev mode first. It may also be useful together with a special set of abbrev definitions for making several global replacements at once.

25.3 Examining and Altering Abbrevs

`M-x list-abbrevs`
: Print a list of all abbrev definitions.

`M-x edit-abbrevs`
: Edit a list of abbrevs; you can add, alter or remove definitions.

The output from M-x list-abbrevs looks like this (after removing blank lines of no significance):

```
(lisp-mode-abbrev-table)
```

```
    "dk"          0     "define-key"
    (global-abbrev-table)
    "dfn"         0     "definitions"
```

(Some blank lines of no semantic significance, and some other abbrev tables, have been omitted.)

A line containing a name in parentheses is the header for abbrevs in a particular abbrev table; global-abbrev-table contains all the global abbrevs, and the other abbrev tables that arenamed after major modes contain the mode-specific abbrevs.

Within each abbrev table, each nonblank line defines one abbrev. The word at the beginning is the abbrev. The number that appears is the number of times the abbrev has been expanded. Emacs keeps track of this to help you see which abbrevs you actually use, in case you decide to eliminate those that you don't use often. The string at the end of the line is the expansion.

`M-x edit-abbrevs` allows you to add, change or kill abbrev definitions by editing a list of them in an Emacs buffer. The list has the same format described above. The buffer of abbrevs is called *Abbrevs*, and is in Edit-Abbrevs mode. This mode redefines the key `C-x C-s` to install the abbrev definitions as specified in the buffer. The command that does this is edit-abbrevs-redefine. Any abbrevs not described in the buffer are eliminated when this is done.

edit-abbrevs is actually the same as list-abbrevs except that it selects the buffer *Abbrevs* whereas list-abbrevs merely displays it in another window.

25.4 Saving Abbrevs

These commands allow you to keep abbrev definitions between editing sessions.

`M-x write-abbrev-file`
: Write a file describing all defined abbrevs.

`M-x read-abbrev-file`
: Read such a file and define abbrevs as specified there.

`M-x quietly-read-abbrev-file`
: Similar but do not display a message about what is going on.

`M-x define-abbrevs`
: Define abbrevs from buffer.

`M-x insert-abbrevs`
: Insert all abbrevs and their expansions into the buffer.

M-x write-abbrev-file reads a file name using the minibuffer and writes a description of all current abbrev definitions into that file. The text stored in the file looks like the output of M-x list-abbrevs.

M-x read-abbrev-file reads a file name using the minibuffer and reads the file, defining abbrevs according to the contents of the file. M-x quietly-read-abbrev-file is the same except that it does not display a message in the echo area saying that it is doing its work. If an empty argument is given to either of these functions, the file name used is the value of the variable abbrev-file-name, which is by default "~/.abbrev_defs".

These commands are used to save abbrev definitions for use in a later session.

Emacs will offer to save abbrevs automatically if you have changed any of them, whenever it offers to save all files (for C-x s or C-x C-c). This feature can be inhibited by setting the variable save-abbrevs to nil.

The commands M-x insert-abbrevs and M-x define-abbrevs are similar to the previous commands but work on text in an Emacs buffer. M-x insert-abbrevs inserts text into the current buffer before point, describing all current abbrev definitions; m-x define-abbrevs parses the entire current buffer and defines abbrevs accordingly.

26. Editing Pictures

If you want to create a picture made out of text characters (for example, a picture of the division of a register into fields, as a comment in a program), use **edit-picture** to enter Picture mode.

In Picture mode, editing is based on the *quarter-plane* model of text, according to which the text characters lie studded on an area that stretches infinitely far to the left and downward. The concept of the end of a line does not exist in this model; the most you can say is where the last nonblank character on the line is found.

Of course, Emacs really always considers text as a sequence of characters, and lines really do have ends. But in Picture mode most frequently-used keys are rebound to commands that simulate the quarter-plane model of text. They do this by inserting spaces or by converting tabs to spaces.

Most of the basic editing commands of Emacs are redefined by Picture mode to do essentially the same thing but in a quarter-plane way. In addition, Picture mode defines `C-c` as a prefix character for more commands that are specific to Picture mode.

One of these commands, `C-c C-c`, is pretty important. Often a picture is part of a larger file that is usually edited in some other major mode. `M-x edit-picture` records the name of the previous major mode, and then you can use the `C-c C-c` command (**Picture-mode-exit**) to restore that mode. `C-c C-c` also deletes spaces from the ends of lines, unless given a numeric argument.

The commands used in Picture mode all work in other modes (provided the **picture** library is loaded), but are not bound to keys except in Picture mode. Note that the descriptions below talk of moving "one column" and so on, but all the picture mode commands handle numeric arguments if their normal equivalents do.

26.1 Basic Editing in Picture Mode

Most keys do the same thing in Picture mode that they usually do, but do it in a quarter-plane style. For example, `C-f` is rebound to run **Picture-forward-column**, which is defined to move point one column to the right, by inserting a space if necessary, so that the actual end of the line makes no difference. `C-b` is rebound to run **Picture-backward-column**, which always moves point right one column, converting a tab to multiple spaces if necessary. `C-n` and `C-p` are rebound to run **Picture-move-down** and **Picture-move-up**, which can either insert spaces or convert tabs as necessary to make sure that point stays in exactly the same column. `C-e` runs **Picture-end-of-line**, which moves to after the last nonblank character on the line. There is no need to change `C-a`, as the choice of screen model does not affect beginnings of lines.

Insertion of text is adapted to the quarter-plane screen model through the use of Overwrite mode (see section 29.1 [Minor Modes], page 159). Self-inserting characters replace existing text, column by column, rather than pushing existing text to the right. `RET` runs Picture-newline, which just moves to the beginning of the following line so that new text will replace that line.

Deletion and killing of text are replaced with erasure. `DEL` runs Picture-backward-clear-column, and replaces the preceding character with a space rather than removing it. `C-d`, Picture-clear-column, does the same thing in a forward direction. `C-k` runs Picture-clear-line; it really kills the contents of lines, but does not ever remove the newlines from the buffer.

To do actual insertion, you must use special commands. `C-o` (Picture-open-line) still creates a blank line, but does so after the current line; it never splits a line. `C-M-o`, split-line, makes sense in Picture mode, so it is not changed. `C-j` (Picture-duplicate-line) inserts below the current line another line with the same contents.

Real deletion can be done with `C-w`, or with `C-c C-d` (which is defined as delete-char, as `C-d` is in other modes), or with one of the picture rectangle commands (see section 26.3 [Picture Rectangle], page 147).

26.2 Controlling Motion after Insert

Since "self-inserting" characters in Picture mode just overwrite and move point, there is no essential restriction on how point should be moved. Normally point moves right, but you can specify any of the eight orthogonal or diagonal directions for motion after a "self-inserting" character. This is useful for drawing lines in the buffer.

`M-‘`	Move left after insertion (Picture-movement-left).
`M-’`	Move right after insertion (Picture-movement-right).
`M--`	Move up after insertion (Picture-movement-up).
`M-=`	Move down after insertion (Picture-movement-down).
`C-c ‘`	Move up and left ("northwest") after insertion (Picture-movement-nw).
`C-c ’`	Move up and right ("northeast") after insertion (Picture-movement-ne).
`C-c /`	Move down and left ("southwest") after insertion (Picture-movement-sw).
`C-c \`	Move down and right ("southeast) after insertion (Picture-movement-se).

Two motion commands move based on the current Picture insertion direction. `C-c C-f` (Picture-motion) moves in the same direction as motion after "insertion" currently does, while `C-c C-b` (Picture-motion-reverse) moves in the opposite direction.

26.3 Picture Mode Tabs

Two kinds of tab-like action are provided in Picture mode. Context-based tabbing is done with `M-TAB` (Picture-tab-search). With no argument, it moves to a point underneath the next "interesting" character that follows whitespace in the previous nonblank line. "Next" here means "appearing at a horizontal position greater than the one point starts out at". With an argument, as in `C-u M-TAB`, this command moves to the next such interesting character in the current line. `M-TAB` does not change the text; it only moves point. "Interesting" characters are defined by the variable picture-tab-chars, which contains a string whose characters are all considered interesting. Its default value is `"!-~"`.

`TAB` itself runs Picture-tab, which operates based on the current tab stop settings; it is the Picture mode equivalent of tab-to-tab-stop. Normally it just moves point, but with a numeric argument it clears the text that it moves over.

The context-based and tab-stop-based forms of tabbing are brought together by the command `C-c TAB`, Picture-set-tab-stops. This command sets the tab stops to the positions which `M-TAB` would consider significant in the current line. The use of this command, together with `TAB`, can get the effect of context-based tabbing. But `M-TAB` is more convenient in the cases where it is sufficient.

26.4 Picture Mode Rectangle Commands

Picture mode defines commands for working on rectangular pieces of the text in ways that fit with the quarter-plane model. The standard rectangle commands may also be useful (see section 10.4 [Rectangles], page 47).

`C-c C-k`	Clear out the region-rectangle. With argument, kill it.
`C-c k` *r*	Similar but save rectangle contents in register *r* first.
`C-c C-y`	Overwrite last killed rectangle into the buffer, with upper left corner at point. With argument, insert instead.
`C-c y` *r*	Similar, but take the rectangle from register *r*.

The commands `C-c C-k` (**Picture-clear-rectangle**) and `C-c k` (**Picture-clear-rectangle-to-register**) differ from the standard rectangle commands in that they normally clear the rectangle instead of deleting it; this is analogous with the way `C-d` is changed in Picture mode.

However, deletion of rectangles can be useful in Picture mode, so these commands delete the rectangle if given a numeric argument.

The Picture mode commands for yanking rectangles differ from the standard ones in overwriting instead of inserting. This is the same way that Picture mode insertion of other text is different from other modes. `C-c C-y` (**Picture-yank-rectangle**) inserts (by overwriting) the rectangle that was most recently killed, while `C-c y` (**Picture-yank-rectangle-from-register**) does likewise for the rectangle found in a specified register.

27. Miscellaneous Commands

This chapter contains several brief topics that do not fit anywhere else.

27.1 Recursive Editing Levels

A *recursive edit* is a situation in which you are using Emacs commands to perform arbitrary editing while in the middle of another Emacs command. For example, when you type `C-r` inside of a query-replace, you enter a recursive edit in which you can change the current buffer.

Exiting the recursive edit means returning to the unfinished command, which continues execution. For example, exiting the recursive edit requested by `C-r` in query-replace causes query replacing to resume. Exiting is done with `C-M-c` (exit-recursive-edit). In most modes, `C-c` also runs this command.

You can also *abort* the recursive edit. This is like exiting, but the unfinished command is immediately aborted. Use the command `C-]` (abort-recursive-edit) for this. See section 30.1 [Quitting], page 173.

The mode line shows you when you are in a recursive edit, by displaying square brackets around the parentheses that always surround the major and minor mode names. Every window's mode line shows this, in the same way, since being in a recursive edit is true regardless of what buffer is selected.

It is possible to be in recursive edits within recursive edits. For example, after typing `C-r` in a query-replace, you might type a command that entered the debugger. In such circumstances, two or more sets of square brackets appear in the mode line. Exiting the inner recursive edit (such as, with the debugger `c` command) would resume the command where it called the debugger. After the end of this command, you would be able to exit the first recursive edit. Aborting also gets out of only one level of recursive edit; it returns immediately to the command level of the previous recursive edit. So you could immediately abort that one too.

Alternatively, the command `M-x top-level` aborts all levels of recursive edits, returning immediately to the top level command reader.

The text being edited inside the recursive edit need not be the same text that you were editing at top level. It depends on what the recursive edit is for. If the command that invokes the recursive edit selects a different buffer first, that is the buffer you will edit recursively. In any case, you can switch buffers within the recursive edit in the normal manner (as long as the buffer-switching keys have not been rebound). You could probably do all the rest of your editing inside the recursive edit, visiting files and all. But this could have surprising effects (such as stack overflow) from time to time. So remember to exit or abory the recursive edit when you no longer need it.

In general, GNU Emacs tries to avoid using recursive edits. It is usually preferable to allow the user to switch among the possible editing modes in any order he likes. With recursive edits, the only way to get to another state is to go "back" to the state that the recursive edit was invoked from.

27.2 Mail

You can read your mail with Emacs in either of two ways, but they are not yet fully documented. `M-x rmail` invokes a mail-reading program that runs entirely within Emacs. `M-x mh-rmail` invokes an interface from Emacs to the mh mail-reading program. Also, `M-x rnews` runs a program much like Rmail but intended for reading Usenet news. For more information, read the documentation of these functions using `C-h f`.

To send a message in Emacs, you start by typing a command (`C-x m`) to select and initialize the *mail* buffer. Then you edit the text and headers of the message in this buffer, and type another command (`C-c C-c`) to send the message.

`C-x m`	Begin composing a message to send.
`C-c C-s`	Send the message, and leave the *mail* buffer selected.
`C-c C-c`	Send the message and select some other buffer. All the key sequences starting with `C-c` that are documented here are defined this way only in Mail mode.
`C-c t`	Move to the 'To' header field, creating one if there is none.
`C-c s`	Move to the '`Subject`' header field, creating one if there is none.
`C-c c`	Move to the '`CC`' header field, creating one if there is none.
`C-c w`	Insert the file ~/.signature at the end of the message text.
`C-c y`	Yank the selected message from Rmail. This command does nothing unless your command to start sending a message was issued with Rmail.

The command `C-x m` (mail) selects a buffer named *mail* and initializes it with the skeleton of an outgoing message. `C-x 4 m` (mail-other-window) does likewise but selects the *mail* buffer in a different window, leaving the previous current buffer visible.

The line in the buffer that says

```
--Text follows this line--
```

is a special delimiter. Whatever follows it is the text of the message; the headers precede it. The delimiter line itself does not appear in the message actually sent.

Header fields you can use include 'To', 'CC' and 'Subject'. 'To' specifies the main recipients of the message. 'CC' specifies additional recipients; both kinds of recipients receive the message just the same, but the people who see their names in the 'CC' field know that they are simply being shown what is addressed to the people in the 'To' field. The 'Subject' field contains a single line giving the topic or purpose of the message. The header fields look like this:

```
To: rms@mc
CC: mly@mc, rg@oz
Subject: The Emacs Manual
--Text follows this line--
```

'Subject:' can be abbreviated 'S:'.

The major mode used in the *mail* buffer is Mail mode, which is much like Text mode except that the character C-c is redefined to be a prefix character, instead of its usual meaning of exit-recursive-edit. The commands that begin with C-c in Mail mode all have to do specifically with editing or sending the message.

There are two ways to send the message. C-c C-s (mail-send) sends the message and marks the *mail* buffer unmodified, but leaves that buffer selected so that you can modify the message (perhaps with new recipients) and send it again. C-c C-c (mail-send-and-exit) sends and then deletes the window (if there is another window) or switches to another buffer. It puts the *mail* buffer at the lowest priority for automatic reselection, since you are finished with using it. This is the usual way to send the message.

Mail mode provides some other special commands that are useful for editing the headers and text of the message before you send it. There are four commands defined to move point to particular header fields: C-c t (mail-to) to move to the 'To' field, C-c s (mail-subject) for the 'Subject' field, and C-c c (mail-cc) for the 'CC' field.

C-c w (mail-signature) adds a standard piece text at the end of the message to say more about who you are. The text comes from the file .signature in your home directory.

When mail sending is invoked from the Rmail mail reader using an Rmail command, `C-c y` can be used inside the *mail* buffer to insert the text of the message you are replying to. Normally it indents each line of that message four spaces and eliminates most header fields. A numeric argument specifies the number of spaces to indent. An argument of just `C-u` says not to indent at all and not to eliminate anything. `C-c y` always uses the current message from the RMAIL buffer, so you can insert several old messages by selecting one in RMAIL, switching to *mail* and yanking it, then switching back to RMAIL to select another.

Because the mail composition buffer is an ordinary Emacs buffer, you can switch to other buffers while in the middle of composing mail, and switch back later (or never). If you use the `C-x m` command again when you have been composing another message but have not sent it, you are asked to confirm before the old message is erased. If you answer `n`, the *mail* buffer is left selected with its old cortents, so you can finish the old message and send it. `C-u C-x m` is another way to do this. Sending the message marks the *mail* buffer "unmodified", which prevents confirmation when `C-x m` is next used.

28. Narrowing

Narrowing means focusing in on some portion of the buffer, making the rest temporarily invisible and inaccessible. Cancelling the narrowing, and making the entire buffer once again visible, is called *widening*. The amount of narrowing in effect in a buffer at any time is called the buffer's *restriction*.

C-x n	Narrow down to between point and mark.
C-x w	Widen to make the entire buffer visible again.

When you have narrowed down to a part of the buffer, that part appears to be all there is. You can't see the rest, you can't move into it (motion commands won't go outside the visible part), you can't change it in any way. However, it is not gone, and if you save the file all the invisible text will be saved. In addition to sometimes making it easier to concentrate on a single subroutine or paragraph by eliminating clutter, narrowing can be used to restrict the range of operation of a replace command or repeating keyboard macro. The word 'Narrow' appears in the mode line whenever narrowing is in effect.

The primary narrowing command is C-x n (narrow-to-region). It sets the current buffer's restrictions so that the text in the current region remains visible but all text before the region or after the region is invisible. Point and mark do not change.

Because narrowing can easily confuse users who do not understand it, narrow-to-region is normally a disabled command. Attempting to use this command asks for confirmation and gives you the option of enabling it; once you enable the command, confirmation will no longer be required for it. See section 29.4.4 [Disabling], page 169.

The way to undo narrowing is to widen with C-x w (widen). This makes all text in the buffer accessible again.

You can get information on what part of the buffer you are narrowed down to using the C-x = command. See section 4.8 [Position Info], page 21.

28.1 Running Shell Commands from Emacs

Emacs has commands for passing single command lines to inferior shell processes; it can also run a shell interactively with input and output to an Emacs buffer *shell*.

M-!	Run a specified shell command line and display the output.
M-|	Run a specified shell command line with region contents as input; optionally replace the region with the output.
M-x shell	Run a subshell with input and output through an Emacs buffer. You can then give commands interactively.

28.1.1 Single Shell Commands

M-! (shell-command) reads a line of text using the minibuffer and creates an inferior shell to execute the line as a command. Standard input from the command comes from the null device. If the shell command produces any output, the output goes into an Emacs buffer named *Shell Command Output*, which is displayed in another window but not selected. A numeric argument, a in M-1 M-!, directs this command to insert any output into the current buffer. In that case, point is left before the output and the mark is set after the output.

M-| (shell-command-on-region) is like M-! but passes the contents of the region as input to the shell command, instead of no input. If a numeric argument is used, meaning insert output in the current buffer, then the old region is deleted first and the output replaces it as the contents of the region.

Both M-! and M-| use shell-file-name to specify the shell to use. This variable is initialized based on your SHELL environment variable when Emacs is started. If the file name does not specify a directory, the directories in the list exec-path are searched; this list is initialized based on the environment variable PATH when Emacs is started. Your .emacs file can override either or both of these default initializations.

With M-! and M-|, Emacs has to wait until the shell command completes. You can quit with C-g; that terminates the shell command.

28.1.2 Interactive Inferior Shell

To run a subshell interactively, putting its typescript in an Emacs buffer, use M-x shell. This creates (or reuses) a buffer named *shell* and runs a subshell with input coming from and output going to that buffer. That is to say, any "terminal output" from the subshell will go into the buffer, advancing point, and any "terminal input" for the subshell comes from text in the buffer. To give input to the subshell, go to the end of the buffer and type the input, terminated by RET.

Emacs does not wait for the subshell to do anything. You can switch windows or buffers and edit them while the shell is waiting, or while it is running a command. Output from the subshell waits until Emacs has time to process it; this happens whenever Emacs is waiting for keyboard input or for time to elapse.

The file name used to load the subshell is the value of the variable explicit-shell-file-name, if that is non-nil. Otherwise, the ESHELL environment variable is used, or the SHELL environment variable if there is no ESHELL. If no directory is specified, the directories in the list exec-path are searched; see above.

As soon as the subshell is started, it is sent as input the contents of the file ~/.emacs_*shellname*, where *shellname* is the name of the file that the shell was loaded from. For example, if you use csh, the file sent to it is ~/.emacs_csh.

The shell buffer uses Shell mode, which attempts to imitate the usual editing and job control characters present in shells that are not under Emacs. Here is a complete list of the special key bindings of Shell mode:

`RET`	At end of buffer send line as input; otherwise, copy current line to end and send it (send-shell-input). When a line is copied to the end, any text at the beginning of the line that matches the variable shell-prompt-pattern is left out; this variable's value should be a regexp string that matches the prompts that you use in your subshell.
`C-d`	At end of buffer, send end-of-file as input; otherwise, delete a character as usual (delete-char-or-send-eof).
`C-u`	Kill all text that has yet to be sent as input (kill-shell-input).
`C-w`	Kill a word before point (backward-kill-word).
`C-c`	Interrupt the shell or its current subjob if any (interrupt-shell-subjob).
`C-z`	Stop the shell or its current subjob if any (stop-shell-subjob).
`C-\`	Send quit signal to the shell or its current subjob if any (quit-shell-subjob).
`C-x C-k`	Delete last batch of output from shell (kill-output-from-shell).
`C-x C-v`	Scroll top of last batch of output to top of window (show-output-from-shell).

cd, pushd and popd commands given to the inferior shell are watched by Emacs so it can keep the *shell* buffer's default directory the same as the shell's working directory.

28.2 Hardcopy Output

`M-x print-buffer`
: Print hardcopy of current buffer using Unix command `lpr -p`. This makes page headings containing the file name and page number.

`M-x lpr-buffer`
: Print hardcopy of current buffer using Unix command `lpr`. This makes no page headings.

`M-x print-region`
: Like `print-buffer` but prints only the current region.

`M-x lpr-region`
: Like `lpr-buffer` but prints only the current region.

All the hardcopy commands pass extra switches to the `lpr` program based on the value of the variable `lpr-switches`. Its value should be a list of strings, each string a switch.

28.3 Dissociated Press

`M-x dissociated-press` is a command for scrambling a file of text either word by word or character by character. Starting from a buffer of straight English, it produces extremely amusing output. The input comes from the current Emacs buffer. Dissociated Press writes its output in a buffer named `*Dissociation*`, and redisplays that buffer after every couple of lines (approximately) to facilitate reading it.

`dissociated-press` asks every so often whether to continue operating. Answer n to stop it. You can also stop at any time by typing `C-g`. The dissociation output remains in the `*Dissociation*` buffer for you to copy elsewhere if you wish.

Dissociated Press operates by jumping at random from one point in the buffer to another. In order to produce plausible output rather than gibberish, it insists on a certain amount of overlap between the end of one run of consecutive words or characters and the start of the next. That is, if it has just printed out 'president' and then decides to jump to a different point in the file, it might spot the 'ent' in 'pentagon' and continue from there, producing 'presidentagon'. Long sample texts produce the best results.

A positive argument to `M-x Dissociated Press` tells it to operate character by character, and specifies the number of overlap characters. A negative argument tells it to operate word by word and specifies the number of overlap words. In this mode, whole words are treated as the elements to be permuted, rather than characters. No argument is equivalent to an argument of two. For your againformation, the output goes only into the buffer `*Dissociation*`. The buffer you start with is not changed.

Dissociated Press produces nearly the same results as a Markov chain based on a frequency table constructed from the sample text. It is, however, an independent, ignoriginal invention. Dissociated Press techniquitously copies several consecutive characters from the sample between random choices, whereas a Markov chain would choose randomly for each word or character. This makes for more plausible sounding results, and runs faster.

It is a mustatement that too much use of Dissociated Press can be a developediment to your real work. Sometimes to the point of outragedy. And keep dissociwords out of your documentation, if you want it to be well userenced and properbose. Have fun. Your buggestions are welcome.

29. Customization

This chapter talks about various topics relevant to adapting the behavior of Emacs in minor ways.

29.1 Minor Modes

Minor modes are options which you can use or not. For example, Auto Fill mode is a minor mode in which `SPC` breaks lines between words as you type. All the minor modes are independent of each other and of the selected major mode. Most minor modes say in the mode line when they are on; for example, '`Fill`' in the mode line means that Auto Fill mode is on.

Append `-mode` to the name of a minor mode to get the name of a command function that turns the mode on or off. Thus, the command to enable or disable Auto Fill mode is called `M-x auto-fill-mode`. These commands are usually invoked with `M-x`, but you can bind keys to them if you wish. With no argument, the function turns the mode on if it was off and off if it was on. This is known as *toggling*. A positive argument always turns the mode on, and an explicit zero argument or a negative argument always turns it off.

Auto Fill mode allows you to enter filled text without breaking lines explicitly. Emacs inserts newlines as necessary to prevent lines from becoming too long. See section 22.6 [Filling], page 106.

Overwrite mode causes ordinary printing characters to replace existing text instead of shoving it over. It is good for editing pictures. For example, if the point is in front of the '`B`' in '`FOOBAR`', then in Overwrite mode typing a `G` changes it to '`FOOGAR`', instead of making it '`FOOGBAR`' as usual.

Abbrev mode allows you to define abbreviations that automatically expand as you type them. For example, '`amd`' might expand to '`abbrev mode`'. See chapter 25 [Abbrevs], page 139, for full information.

29.2 Variables

A *variable* is a Lisp symbol which has a value. The symbol's name is also called the name of the variable. Variable names can contain any characters, but conventionally they are chosen to be words separated by hyphens. A variable can have a documentation string which describes what kind of value it should have and how the value will be used.

Lisp allows any variable to have any kind of value, but most variables that Emacs uses require a value of a certain type. Often the value should always be a string, or should always be a number. Sometimes we say that a certain feature is turned on if a variable is "non-nil," meaning that if the variable's value is nil, the feature is off, but the feature is on for *any* other value. The conventional value to use to turn on the feature—since you have to pick one particular value when you set the variable—is t.

Emacs uses many Lisp variables for internal recordkeeping, as any Lisp program must, but the most interesting variables for you are the ones that exist for the sake of customization. Emacs does not (usually) change the values of these variables; instead, you set the values, and thereby alter and control the behavior of certain Emacs commands. These variables are called *options*. Most options are documented in this manual, and appear in the Variables Index.

One example of a variable which is an option is fill-column, which specifies the position of the right margin (as a number of characters from the left margin) to be used by the fill commands (see section 22.6 [Filling], page 106).

`C-h v`
`M-x describe-variable`
: Print the value and documentation of a variable.

`M-x set-variable`
: Change the value of a variable.

`M-x make-local-variable`
: Make a variable have a local value in the current buffer.

`M-x kill-local-variable`
: Make a variable use its global value in the current buffer.

`M-x list-options`
: Display a buffer listing names, values and documentation of all options.

`M-x edit-options`
: Change option values by editing a list of options.

29.2.1 Examining and Setting Variables

To examine the value of a single variable, use `C-h v` (describe-variable), which reads a variable name using the minibuffer, with completion. It prints both the value and the documentation of the variable.

```
C-h v fill-column RET
```

prints something like

```
fill-column's value is 75

Documentation:
*Column beyond which automatic line-wrapping should happen.
Separate value in each buffer.
```

The star at the beginning of the documentation indicates that this variable is an option. Commands that operate on a single, specified variable are not restricted to options; they allow any variable name.

`M-x list-options` displays a list of all Emacs option variables, in an Emacs buffer named ***List Options***. Each option is shown with its documentation and its current value. Here is what a portion of it might look like:

```
;; exec-path:
("." "/usr/local/bin" "/usr/ucb" "/bin" "/usr/bin" "/u2/emacs/etc")
*List of directories to search programs to run in subprocesses.
Each element is a string (directory name) or nil (try default directory).
;;
;; fill-column:
75
*Column beyond which automatic line-wrapping should happen.
Separate value in each buffer.
;;
;; find-file-hook:
nil
*If non-nil specifies a function to be called after a buffer
is found or reverted from a file.
The buffer's local variables (if any) will have been processed before the
function is called.
;;
```

`M-x edit-options` goes one step farther and selects the ***List Options*** buffer; this buffer uses the major mode Options mode, which provides commands that allow you to point at an option and change its value:

`s`	Set the variable point is in or near to a new value read using the minibuffer.
`x`	Toggle the variable point is in or near: if the value was **nil**, it becomes **t**; otherwise it

	becomes nil.
1	Set the variable point is in or near to t.
0	Set the variable point is in or near to nil.
n	
p	Move to the next or previous variable.

If you know which option you want to set, you set it without waiting for edit-options to make the option list using `M-x set-variable`. This reads the variable name with the minibuffer (with completion), and then reads a Lisp expression for the new value using the minibuffer a second time. For example,

```
M-x set-variable RET fill-column RET 75 RET
```

sets fill-column to 75.

If you want to set a variable a particular way each time you use Emacs, you can use the Lisp function setq in your .emacs file.

29.2.2 Local Variables

Any variable can be made *local* to a specific Emacs buffer. This means that its value in that buffer is independent of its value in other buffers. A few variables are always local in every buffer; aside from this, major modes always make the variables they set local to the buffer. This is why changing major modes in one buffer has no effect on other buffers. Every other Emacs variable has a *global* value which is in effect in all buffers that have not made the variable local.

`M-x make-local-variable` reads the name of a variable and makes it local to the current buffer. Further changes in this buffer will not affect others, and further changes in the global value will not affect this buffer.

`M-x kill-local-variable` reads the name of a variable and makes it cease to be local to the current buffer. The global value of the variable henceforth is in effect in this buffer. Setting the major mode kills all the local variables of the buffer, except for those variables that are always local to every buffer.

29.2.3 Local Variables in Files

A file can contain a *local variables list*, which specifies the values to use for certain Emacs variables when that file is edited. Visiting the file checks for a local variables list and makes each variable in the list local to the buffer in which the file is visited, with the value specified in the file.

A local variables list goes near the end of the file, in the last page. (It is often best to put it on a page by itself.) The local variables list starts with a line containing the string '`Local Variables:`', and ends with a line containing the string '`End:`'. In between come the variable names and values, one set per line, as '*variable*: *value*'. The *values* are not evaluated; they are used literally.

The line which starts the local variables list does not have to say just '`Local Variables:`'. If there is other text before '`Local Variables:`', that text is called the *prefix*, and if there is other text after, that is called the *suffix*. If these are present, each entry in the local variables list should have the prefix before it and the suffix after it. This includes the '`End:`' line. The prefix and suffix are included to disguise the local variables list as a comment so that the compiler or text formatter will not be perplexed by it. If you do not need to disguise the local variables list as a comment in this way, do not bother with a prefix or a suffix.

Two "variable" names are special in a local variables list: a value for the variable mode really sets the major mode, and a value for the variable eval is simply evaluated as an expression and the value is ignored. These are not real variables; setting such variables in any other context has no such effect. If mode is used in a local variables list, it should be the first entry in the list.

Here is an example of a local variables list:

```
;;; Local Variables: ***
;;; mode:lisp ***
;;; comment-column:0 ***
;;; comment-start: ";;; "  ***
;;; comment-end:"***" ***
;;; End: ***
```

Note that the prefix is '`;;; `' and the suffix is '` ***`'. Note also that comments in the file begin with and end with the same strings. Presumably the file contains code in a language which is like Lisp (like it enough for Lisp mode to be useful) but in which comments start and end in that way. The prefix and suffix are used in the local variables list to make the list appear as comments when the file is read by the compiler or interpreter for that language.

The start of the local variables list must be no more than 3000 characters from the end of the file, and must be in the last page if the file is divided into pages. Otherwise, Emacs will not notice it is there. The purpose of this is so that a stray '`Local Variables:`' not in the last page does not confuse Emacs, and so that visiting a long file that is all one page and has no local variables list need not take the time to search the whole file.

29.3 Keyboard Macros

A *keyboard macro* is a command defined by the user to abbreviate a sequence of other commands. For example, if you discover that you are about to type `C-n C-d` forty times, you can speed your work by defining a keyboard macro to do `C-n C-d` and calling it with a repeat count of forty.

`C-x (`	Start defining a keyboard macro.
`C-x )`	End the definition of a keyboard macro.
`C-x e`	Execute the most recent keyboard macro.
`C-u C-x (`	Re-execute last keyboard macro, then add more commands to its definition.
`C-x q`	Ask for confirmation when the keyboard macro is executed.
`M-x name-last-kbd-macro`	Give a permanent name (for the duration of the session) to the most recently defined keyboard macro.
`M-x write-kbd-macro`	Store the definition of a keyboard macro into a file.
`M-x append-kbd-macro`	Append the definition of a keyboard macro to the end of a file.

Keyboard macros differ from ordinary Emacs commands in that they are written in the Emacs command language rather than in Lisp. This makes it easier for the novice to write them, and makes them more convenient as temporary hacks. However, the Emacs command language is not powerful enough as a programming language to be useful for writing anything intelligent or general. For such things, Lisp must be used.

You define a keyboard macro while executing the commands which are the definition. Put differently, as you are defining a keyboard macro, the definition is being executed for the first time. This way, you can see what the effects of your commands are, so that you don't have to figure them out in your head. When you are finished, the keyboard macro is defined and also has been, in effect, executed once. You can then do the whole thing over again by invoking the macro.

29.3.1 Basic Use

To start defining a keyboard macro, type the `C-x (` command (**start-kbd-macro**). From then on, your commands continue to be executed, but also become part of the definition of the macro. '`Def`' appears in the mode line to remind you of what is going on. When you are finished, the `C-x )` command (**end-kbd-macro**) terminates the definition (without becoming part of it!). For example

```
C-x ( M-F foo C-x )
```

defines a macro to move forward a word and then insert '`foo`'.

The macro thus defined can be invoked again with the `C-x e` command (**call-last-kbd-macro**), which may be given a repeat count as a numeric argument to execute the macro many times. `C-x )` can also be given a repeat count as an argument, in which case it repeats the macro that many times right after defining it, but defining the macro counts as the first repetition (since it is executed as you define it). So, giving `C-x )` an argument of 4 executes the macro immediately 3 additional times. An argument of zero to `C-x e` or `C-x )` means repeat the macro indefinitely (until it gets an error, or you type `C-g`).

If you wish to repeat an operation at regularly spaced places in the text, define a macro and include as part of the macro the commands to move to the next place you want to use it. For example, if you want to change each line, you should position point at the start of a line, and define a macro to change that line and leave point at the start of the next line. Then repeating the macro will operate on successive lines.

After you have terminated the definition of a keyboard macro, you can add to the end of its definition by typing `C-U C-x (`. This is equivalent to plain `C-x (` followed by retyping the whole definition so far. As a consequence it re-executes the macro as previously defined.

29.3.2 Naming and Installing Keyboard Macros

If you wish to save a keyboard macro for longer than until you define the next one, you must give it a name or install it on a command sequence. To give the macro a name, use `M-x name-last-kbd-macro`. This reads a name as an argument using the minibuffer and defines that name to execute the macro. The macro name is a Lisp symbol, and defining it in this way makes it valid for calling with `M-x` or for binding a key to it with **global-set-key** (see section 29.4 [Keymaps], page 166).

29.3.3 Executing Macros with Variations

Using `C-x q` (kbd-macro-query), you can get an effect similar to that of query-replace, where the macro asks you each time around whether to make a change. When you are defining the macro, type `C-x q` at the point where you want the query to occur. During macro definition, the `C-x q` does nothing, but when the macro is invoked the `C-x q` reads a character from the terminal to decide whether to continue.

The special answers are SPC, DEL, `C-d`, `C-l` and `C-r`. Any other character terminates execution of the keyboard macro and is then read as a command. SPC means to continue. DEL means to skip the remainder of this repetition of the macro, starting again from the beginning in the next repetition. `C-d` means to skip the remainder of this repetition and cancel further repetition. `C-l` clears the screen and asks you again for a character to say what to do. `C-r` enters a recursive editing level, in which you can perform editing which is not part of the macro. When you exit the recursive edit using `C-M-c`, you are asked again how to continue with the keyboard macro. If you type a SPC at this time, the rest of the macro definition is executed. It is up to you to leave point and the text in a state such that the rest of the macro will do what you want.

`C-u C-x q`, `C-x q` with a numeric argument, performs a different function. It enters a recursive edit reading input from the keyboard, both when you type it during the definition of the macro, and when it is executed from the macro. During definition, the editing you do inside the recursive edit does not become part of the macro. During macro execution, the recursive edit gives you a chance to do some particularized editing. See section 27.1 [Recursive Edit], page 149.

29.4 Customizing Key Bindings

This section deals with the *keymaps* which define the bindings between keys and functions, and say how you can customize these bindings.

29.4.1 Commands and Functions

A command is a Lisp function whose definition says how to call it interactively (from the editor command loop). Like every Lisp function, a command has a function name, a Lisp symbol whose name usually consists of lower case letters and dashes.

29.4.2 Keymaps

The bindings between characters and command functions are recorded in data structures called *keymaps*. Emacs has many of these. One, the *global* keymap, defines the meanings of the single keys that are defined regardless of major mode. It is the value of the variable **global-map**.

Each major mode has another keymap, its *local keymap*, which contains overriding definitions for the single keys that are to be redefined in that mode. Each buffer records which local keymap is installed for it at any time, and the current buffer's local keymap is the only one that directly affects command execution. The local keymaps for Lisp mode, C mode, and other major modes exist always even when not in use. They are the values of the variables **lisp-mode-map**, **c-mode-map**, and so on. For major modes less often used, the local keymap is often constructed only when the mode is used for the first time in a session. This is to save space.

Finally, each prefix key has a keymap which defines the key sequences that start with it. For example, **ctl-x-map** is the keymap used for characters following a `C-x`, and **help-map** is the keymap used for characters following a `C-h`. **esc-map** is the keymap used for characters following `ESC`, and therefore for all Meta characters (see below). In fact, the definition of a prefix key is just the keymap to use for looking up the following character. Actually, the definition is usually a Lisp symbol whose function definition is the following character keymap. The effect is the same, but it provides a command name for the prefix key that can be used as a description of what the prefix key is for. Thus, the binding of `C-x` is the symbol **Ctl-X-Prefix**, whose function definition is the keymap for `C-x` commands, the value of **ctl-x-map**.

Prefix key definitions of this sort can appear in either the global map or a local map. The definitions of `C-x`, `C-h` and `ESC` as prefix keys appear in the global map, so these prefix keys are always available. Modes such as Mail mode and Picture mode that make `C-c` into a prefix character do so by putting prefix definitions into their local maps. A mode can also put a prefix definition of a global prefix character such as `C-x` into its local map. This is how major modes override the definitions of certain keys that start with `C-x`. When both the global and local definitions of a key are other keymaps, the next character is looked up in both keymaps, with the local definition overriding the global one as usual. So, the character after the `C-x` is looked up in both the major mode's own keymap for redefined `C-x` commands and in **ctl-x-map**. If the major mode's own keymap for `C-x` commands contains **nil**, the definition from the global keymap for `C-x` commands is used.

A keymap is actually a Lisp object. The simplest form of keymap is a Lisp vector of length 128. The binding for a character in such a keymap is found by indexing into the vector with the character as an index. A keymap can also be a Lisp list whose car is the symbol **keymap** and whose remaining elements are pairs of the form (*char* . *binding*). Such lists are called *sparse keymaps* because they are used when most of the characters' entries will be **nil**. Sparse keymaps are used mainly for prefix characters.

Keymaps are only of length 128, so what about Meta characters, whose codes are from 128 to 255? A key that contains a Meta character actually represents it as a sequence of two characters, the first of which is ESC. So the key M-a is really represented as ESC a, and its binding is found at the slot for 'a' in esc-map.

29.4.3 Changing Key Bindings Interactively

The way to redefine an Emacs command is to change an entry in a keymap. You can change the global keymap, in which case the change is effective in all major modes (except those that have their own overriding local definitions for the same key). Or you can change the current buffer's local map, which affects all buffers using the same major mode.

M-x **global-set-key** RET *key cmd* RET
: Defines *key* globally to run *cmd*.

M-x **local-set-key** RET *key cmd* RET
: Defines *key* locally (in the major mode now in effect) to run *cmd*.

For example,

```
M-x global-set-key RET C-f next-line RET
```

would redefine C-f to move down a line. The fact that *cmd* is read second makes it serve as a kind of confirmation for *key*.

There is no way to specify a particular prefix keymap as the one to redefine in, but that is not necessary, as you can include prefixes in *key*. *key* is read by reading characters one by one until they amount to a complete key (that is, not a prefix key). Thus, if you type C-f for *key*, that's the end; the minibuffer is entered immediately to read *cmd*. But if you type C-x, another character is read; if that is 4, another character is read, and so on. For example,

```
M-x global-set-key RET C-x 4 $ dictionary-other-window RET
```

would redefine C-x 4 $ to run the (fictitious) command dictionary-other-window.

29.4.4 Disabling Commands

Disabling a command marks the command as requiring confirmation before it can be executed. The purpose of disabling a command is to prevent beginning users from executing it by accident and being confused.

The direct mechanism for disabling a command is to have a non-nil disabled property on the Lisp symbol for the command. These properties are normally set up by the user's .emacs file with Lisp expressions such as

```
    (put 'delete-region 'disabled t)
```

You can make a command disabled either by editing the .emacs file directly or with the command `M-x disable-command`, which edits the .emacs file for you. To cancel the disablement of a command, use `M-x enable-command`.

Attempting to invoke a disabled command interactively in Emacs causes the display of a window containing the command's name, its documentation, and some instructions on what to do immediately; then Emacs asks for input saying whether to execute the command as requested, enable it and execute, or cancel it. If you decide to enable the command, you is asked whether to do this permanently (that is, change your .emacs file) or just for the current session.

Whether a command is disabled is independent of what key is used to invoke it; it also applies if the command is invoked using `M-x`. Disabling a command has no effect on calling it as a function from Lisp programs.

29.5 The Syntax Table

All the Emacs commands which parse words or balance parentheses are controlled by the *syntax table*. The syntax table says which characters are opening delimiters, which are parts of words, which are string quotes, and so on. Actually, each major mode has its own syntax table (though sometimes related major modes use the same one) which it installs in each buffer that uses that major mode. The syntax table installed in the current buffer is the one that all commands use. So we will call it "the syntax table". A syntax table is a Lisp object, a vector of length 256 whose elements are numbers.

The syntax table entry for a character holds six pieces of information:

- The syntactic class of the character, represented as a small integer.

- The matching delimiter, for delimiter characters only. The matching delimiter of '(' is ')', and vice versa.
- A flag saying whether the character is the first character of a two-character comment starting sequence.
- A flag saying whether the character is the second character of a two-character comment starting sequence.
- A flag saying whether the character is the first character of a two-character comment ending sequence.
- A flag saying whether the character is the second character of a two-character comment ending sequence.

The syntactic classes are stored internally as small integers, but are usually described to or by the user with characters. For example, '(' is used to specify the syntactic class of opening delimiters. Here is a table of syntactic classes, with the characters that specify them.

`SPC`	The class of whitespace characters.
`w`	The class of word-constituent characters.
`_`	The class of characters that are part of symbol names but not words. This class is represented by '_' because the character '_' has this class in both C and Lisp.
`(`	The class of opening delimiters.
`)`	The class of closing delimiters.
`'`	The class of expression-adhering characters. These characters are part of a symbol if found within or adjacent to one one, and are part of a following expression if immediately preceding one, but are like whitespace if surrounded by whitespace.
`"`	The class of string-quote characters. They match each other in pairs, and the characters within the pair all lose their syntactic significance except for the '\' and '/' classes of escape characters, which can be used to include a string-quote inside the string.
`$`	The class of self-matching delimiters. This is intended for TEX's '$', but it has not really been well worked out what it should do. If you don't like its current behavior, please complain.
`/`	The class of escape characters that always just deny the following character its special syntactic significance. The character after one of these escapes is always treated as alphabetic.
`\`	The class of C-style escape characters. In practice, these are treated just like '/'-class characters, because the extra possibilities for C escapes (such as being followed by digits) have no effect on where the containing expression ends.
`<`	The class of comment-starting characters. Only single-character comment starters (such as ';' in Lisp mode) are represented this way.

>	The class of comment-ending characters. Newline has this syntax in Lisp mode.

The characters marked as part of two-character comment delimiters can have other syntactic functions most of the time. The comment-delimiter significance overrides when the pair of characters occur together in the proper order. Only the list and sexp commands use the syntax table to find comments; the commands specifically for comments have other variables that tell them where to find comments. And the list and sexp commands notice comments only if parse-sexp-ignore-comments is non-nil. This variable is set to nil in modes where comment-terminator sequences are liable to appear where there is no comment; for example, in Lisp mode where the comment terminator is a newline but not every newline ends a comment.

`M-x modify-syntax-entry` is the command to change a character's syntax in the current syntax table. It can be used interactively, and is also the means used by major modes to initialize their own syntax tables. Its first argument is the character to change. The second argument is a string that specifies the new syntax:

1. The first character in the string specifies the syntactic class. It is one of the characters in the previous table.
2. The second character is the matching delimiter. For a character that is not an opening or closing delimiter, this should be a space, or may be omitted if no following characters are needed.
3. The remaining characters are flags. The flag characters allowed are

1	Mark this character as the first of a two-character comment-start sequence.
2	Mark this character as the second of a two-character comment-start sequence.
3	Mark this character as the first of a two-character comment-end sequence.
4	Mark this character as the second of a two-character comment-end sequence.

A description of the contents of the current syntax table can be displayed with `C-h s` (describe-syntax). The description of each character includes both the string you would have to give to modify-syntax-entry to set up that character's current syntax, and some English to explain that string if necessary.

30. Correcting Mistakes and Emacs Problems

If you type an Emacs command you did not intend, the results are often mysterious. This chapter tells what you can do to cancel your mistake or recover from a mysterious situation. Emacs bugs and system crashes are also considered.

30.1 Quitting and Aborting

`C-g`	Quit. Cancel running or partially typed command.
`C-]`	Abort recursive editing level and cancel the command which invoked it.
`M-x top-level`	Abort all recursive editing levels that are currently executing.

There are three ways of cancelling commands which are not finished executing: *quitting* with `C-g`, and *aborting* with `C-]` or `M-x top-level`. Quitting is cancelling a partially typed command or one which is already running. Aborting is getting out of a recursive editing level and cancelling the command that invoked the recursive edit.

Quitting with `C-g` is used for getting rid of a partially typed command, or a numeric argument that you don't want. It also stops a running command in the middle in a relatively safe way, so you can use it if you accidentally give a command which takes a long time. In particular, it is safe to quit out of killing; either your text will *all* still be there, or it will *all* be in the kill ring (or maybe both). Quitting an incremental search does special things documented under searching; in general, it may take two successive `C-g` characters to get out of a search. `C-g` works by setting the variable quit-flag to t the instant `C-g` is typed; Emacs Lisp checks this variable frequently and quits if it is non-nil. `C-g` is only actually executed as a command if it is typed while Emacs is waiting for input.

If you quit twice in a row before the first `C-g` is recognized, you activate the "emergency escape" feature and return to the shell. See section 30.2.5 [Emergency Escape], page 175.

Aborting with `C-]` (abort-recursive-edit) is used to get out of a recursive editing level and cancel the command which invoked it. Quitting with `C-g` does not do this, and could not do this, because it is used to cancel a partially typed command *within* the recursive editing level. Both operations are useful. For example, if you are in the Emacs debugger (see section 24.5 [Lisp Debug], page 135) and have typed `C-u 8` to enter a numeric argument, you can cancel that argument with `C-g` and remain in the debugger.

The command `M-x top-level` is equivalent to "enough" `C-]` commands to get you out of all the levels of subsystems and recursive edits that you are in. `C-]` gets you out one level at a time, but `M-x top-level` goes out all levels at once. Both `C-]` and `M-x top-level` are like all other commands, and unlike `C-g`, in that they are effective only when Emacs is ready for a command. `C-]` is an ordinary key and has its meaning only because of its binding in the keymap.

30.2 Dealing with Emacs Trouble

This section describes various conditions which can cause Emacs not to work, or cause it to display strange things, and how you can correct them.

30.2.1 Recursive Editing Levels

Recursive editing levels are important and useful features of Emacs, but they can seem like malfunctions to the user who does not understand them.

If the mode line starts with a bracket '`[`', you have entered a recursive editing level. To get back to top level, type `M-x top-level`. See section 27.1 [Recursive Edit], page 149.

30.2.2 Garbage on the Screen

If the data on the screen looks wrong, the first thing to do is see whether the text is really wrong. Type `C-l`, to redisplay the entire screen. If it appears correct after this, the problem was entirely in the previous screen update.

Display updating problems often result from an incorrect termcap entry for the terminal you are using. The file etc/TERMS gives the fixes for known problems of this sort. INSTALL contains general advice for these problems in one of its sections. Very likely there is simply insufficient padding for certain display operations. To investigate the possibility that you have this sort of problem, try Emacs on another terminal made by a different manufacturer. If problems happen frequently on one kind of terminal but not another kind, it is likely to be a bad termcap entry, though it could also be due to a bug in Emacs that appears for terminals that have or that lack specific features.

30.2.3 Garbage in the Text

If `C-l` shows that the text is wrong, try undoing the changes to it using `C-x u` until it gets back to a state you consider correct. Also try `C-h l` to find out what command you typed to produce the observed results.

30.2.4 Spontaneous Entry to Incremental Search

If Emacs spontaneously displays '`I-search:`' at the bottom of the screen, it means that the terminal is sending `C-s` and `C-q` according to the badly designed xon/xoff "flow control" protocol. You should try to prevent this by putting the terminal in a mode where it will not use flow control or giving it enough padding that it will never send a `C-s`. If that cannot be done, you must tell Emacs to expect flow control to be used, until you can get a properly designed terminal.

Information on how to do these things can be found in the file INSTALL in the Emacs distribution.

30.2.5 Emergency Escape

Because at times there have been bugs causing Emacs to loop without checking quit-flag, a special feature causes Emacs to be suspended immediately if you type a second `C-g` while the flag is already set. So you can always get out of GNU Emacs. Normally Emacs recognizes and clears quit-flag (and quits!) quickly enough to prevent this from happening.

When you resume Emacs after a suspension caused by multiple `C-g`, it asks two questions before going back to what it had been doing:

```
Checkpoint?
Dump core?
```

Answer each one with `y` or `n` followed by RET.

Saying `y` to '`Checkpoint?`' causes immediate auto-saving of all modified buffers in which auto-saving is enabled.

Saying `y` to '`Dump core?`' causes an illegal instruction to be executed, dumping core. This is to enable a wizard to figure out why Emacs was failing to quit in the first place. Execution does not really continue after a core dump. If you answer `n`, execution does continue. With luck, GNU

Emacs will ultimately check **quit-flag** and quit normally. If not, and you type another `C-g`, it is suspended again.

If Emacs is not really hung, just slow, you may invoke the double `C-g` feature without really meaning to. Then just resume and answer `n` to both questions, and you will arrive at your former state. Presumably the quit you requested will happen soon.

30.3 Reporting Bugs

Sometimes you will encounter a bug in Emacs. To get it fixed, you must report it. It is your duty to do so; but you must know when to do so and how if it is to be useful.

30.3.1 When Is There a Bug

If Emacs executes an illegal instruction, or dies with an operating system error message that indicates a problem in the program (as opposed to "disk full"), then it is certainly a bug.

If Emacs updates the display in a way that does not correspond to what is in the buffer, then it is certainly a bug. If a command seems to do the wrong thing but the problem corrects itself if you type `C-l`, it is a case of incorrect display updating.

Taking forever to complete a command can be a bug, but you must make certain that it was really Emacs's fault. Some commands simply take a long time. Type `C-g` and then `C-h l` to see whether the input Emacs received was what you intended to type; if the input was such that you *know* it should have been processed quickly, report a bug. If you don't know whether the command should take a long time, find out by looking in the manual or by asking for assistance.

If a command you are familiar with causes an Emacs error message in a case where its usual definition ought to be reasonable, it is probably a bug.

If a command does the wrong thing, that is a bug. But be sure you know for certain what it ought to have done. If you aren't familiar with the command, or don't know for certain how the command is supposed to work, then it might actually be working right. Rather than jumping to conclusions, show the problem to someone who knows for certain.

Finally, a command's intended definition may not be best for editing with. This is a very important sort of problem, but it is also a matter of judgment. Also, it is easy to come to such a conclusion out of ignorance of some of the existing features. It is probably best not to complain about such a problem until you have checked the documentation in the usual ways, feel confident that you understand it, and know for certain that what you want is not available. If you are not sure what the command is supposed to do after a careful reading of the manual, check the index and glossary for any terms that may be unclear. If you still do not understand, this indicates a bug in the manual. The manual's job is to make everything clear. It is just as important to report documentation bugs as program bugs.

If the on-line documentation string of a function or variable disagrees with the manual, one of them must be wrong, so report the bug.

30.3.2 How to Report a Bug

When you decide that there is a bug, it is important to report it and to report it in a way which is useful. What is most useful is an exact description of what commands you type, starting with the shell command to run Emacs, until the problem happens. Always include the version number of Emacs that you are using; type `M-x emacs-version` to print this.

The most important principle in reporting a bug is to report *facts*, not hypotheses or categorizations. It is always easier to report the facts, but people seem to prefer to strain to posit explanations and report them instead. If the explanations are based on guesses about how Emacs is implemented, they will be useless; we will have to try to figure out what the facts must have been to lead to such speculations. Sometimes this is impossible. But in any case, it is unnecessary work for us.

For example, suppose that you type `C-x C-f /glorp/baz.ugh RET`, visiting a file which (you know) happens to be rather large, and Emacs prints out '`I feel pretty today`'. The best way to report the bug is with a sentence like the preceding one, because it gives all the facts and nothing but the facts.

Do not assume that the problem is due to the size of the file and say, "When I visit a large file, Emacs prints out '`I feel pretty today`'." This is what we mean by "guessing explanations". The problem is just as likely to be due to the fact that there is a `z` in the file name. If this is so, then when we got your report, we would try out the problem with some "large file", probably with no `z` in its name, and not find anything wrong. There is no way in the world that we could guess that we should try visiting a file with a `z` in its name.

Alternatively, the problem might be due to the fact that the file starts with exactly 25 spaces. For this reason, you should make sure that you inform us of the exact contents of any file that is needed to reproduce the bug. What if the problem only occurs when you have typed the `C-x C-a` command previously? This is why we ask you to give the exact sequence of characters you typed since starting to use Emacs.

You should not even say "visit a file" instead of `C-x C-f` unless you *know* that it makes no difference which visiting command is used. Similarly, rather than saying "if I have three characters on the line," say "after I type **RET A B C RET** `C-p`," if that is the way you entered the text.

If you are not in Fundamental mode when the problem occurs, you should say what mode you are in.

Check whether any programs you have loaded into the Lisp world, including your .emacs file, set any variables that may affect the functioning of Emacs. Also, see whether the problem happens in a freshly started Emacs without loading your .emacs file (start Emacs with the -q switch to prevent loading the init file.) If the problem does *not* occur then, it is essential that we know the contents of any programs that you must load into the Lisp world in order to cause the problem to occur.

If the problem does depend on an init file or other Lisp programs that are not part of the Lisp Machine system, then you should make sure it is not a bug in those programs by complaining to their maintainers, first. After they verify that they are using Emacs in a way that is supposed to work, they should report the bug.

If you can tell us a way to cause the problem without visiting any files, please do so. This makes it much easier to debug. If you do need files, make sure you arrange for us to see their exact contents. For example, it can often matter whether there are spaces at the ends of lines, or a newline after the last line in the buffer (nothing ought to care whether the last line is terminated, but tell that to the bugs).

The easy way to record the input to Emacs precisely is to to write a dribble file; execute the Lisp expression

```
(open-dribble-file "~/dribble")
```

using `Meta-ESC` or from the `*scratch*` buffer just after starting Emacs. From then on, all Emacs input will be written in the specified dribble file until the Emacs process is killed.

For possible display bugs, it is important to report the terminal type (the value of environment variable TERM), the termcap entry for the terminal (since /etc/termcap is not identical on all machines), and the output that Emacs actually sent to the terminal. The way to collect this output is to execute the Lisp expression

```
    (open-termscript "~/termscript")
```

using `Meta-ESC` or from the `*scratch*` buffer just after starting Emacs. From then on, all output from Emacs to the terminal will be written in the specified termscript file as well, until the Emacs process is killed. If the problem happens when Emacs starts up, put this expression into your ~/.emacs file so that the termscript file will be open when Emacs displays the screen for the first time.

The address for reporting bugs is

GNU Emacs Bugs
545 Tech Sq, rm 703
Cambridge, MA 02139

or, on Usenet, mail to '`eddie!bug-gnu-emacs%prep`'.

The GNU Manifesto

What's GNU? Gnu's Not Unix!

GNU, which stands for Gnu's Not Unix, is the name for the complete Unix-compatible software system which I am writing so that I can give it away free to everyone who can use it. Several other volunteers are helping me. Contributions of time, money, programs and equipment are greatly needed.

So far we have a portable C and Pascal compiler which compiles for Vax and 68000 (though needing much rewriting), an Emacs-like text editor with Lisp for writing editor commands, a yacc-compatible parser generator, a linker, and around 35 utilities. A shell (command interpreter) is nearly completed. When the kernel and a debugger are written, by the end of 1985 I hope, it will be possible to distribute a GNU system suitable for program development. After this we will add a text formatter, an Empire game, a spreadsheet, and hundreds of other things, plus on-line documentation. We hope to supply, eventually, everything useful that normally comes with a Unix system, and more.

GNU will be able to run Unix programs, but will not be identical to Unix. We will make all improvements that are convenient, based on our experience with other operating systems. In particular, we plan to have longer filenames, file version numbers, a crashproof file system, filename completion perhaps, terminal-independent display support, and eventually a Lisp-based window system through which several Lisp programs and ordinary Unix programs can share a screen. Both C and Lisp will be available as system programming languages. We will try to support UUCP, MIT Chaosnet, and Internet protocols for communication.

GNU is aimed initially at machines in the 68000/16000 class, with virtual memory, because they are the easiest machines to make it run on. The extra effort to make it run on smaller machines will be left to someone who wants to use it on them.

Why I Must Write GNU

I consider that the golden rule requires that if I like a program I must share it with other people who like it. Software sellers want to divide the users and conquer them, making each user agree not to share with others. I refuse to break solidarity with other users in this way. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. For years I worked within the Artificial Intelligence Lab to resist such tendencies and other inhospitalities, but now they have gone too far: I cannot remain in an institution where such things are done for me against my will.

So that I can continue to use computers without dishonor, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free. I have resigned from the AI lab to deny MIT any legal excuse to prevent me from giving GNU away.

Why GNU Will Be Compatible with Unix

Unix is not my ideal system, but it is not too bad. The essential features of Unix seem to be good ones, and I think I can fill in what Unix lacks without spoiling them. And a system compatible with Unix would be convenient for many other people to adopt.

How GNU Will Be Available

GNU is not in the public domain. Everyone will be permitted to modify and redistribute GNU, but no distributor will be allowed to restrict its further redistribution. That is to say, proprietary modifications will not be allowed. I want to make sure that all versions of GNU remain free.

Why Many Other Programmers Want to Help

I have found many other programmers who are excited about GNU and want to help.

Many programmers are unhappy about the commercialization of system software. It may enable them to make more money, but it requires them to feel in conflict with other programmers in general rather than feel as comrades. The fundamental act of friendship among programmers is the sharing of programs; marketing arrangements now typically used essentially forbid programmers to treat others as friends. The purchaser of software must choose between friendship and obeying the law. Naturally, many decide that friendship is more important. But those who believe in law often do not feel at ease with either choice. They become cynical and think that programming is just a way of making money.

By working on and using GNU rather than proprietary programs, we can be hospitable to everyone and obey the law. In addition, GNU serves as an example to inspire and a banner to rally others to join us in sharing. This can give us a feeling of harmony which is impossible if we use software that is not free. For about half the programmers I talk to, this is an important happiness that money cannot replace.

How You Can Contribute

I am asking computer manufacturers for donations of machines and money. I'm asking individuals for donations of programs and work.

One consequence you can expect if you donate machines is that GNU will run on them at an early date. The machines should be complete, ready to use systems, approved for use in a residential area, and not in need of sophisticated cooling or power.

I have found very many programmers eager to contribute part-time work for GNU. For most projects, such part-time distributed work would be very hard to coordinate; the independently-written parts would not work together. But for the particular task of replacing Unix, this problem is absent. A complete Unix system contains hundreds of utility programs, each of which is documented separately. Most interface specifications are fixed by Unix compatibility. If each contributor can write a compatible replacement for a single Unix utility, and make it work properly in place of the original on a Unix system, then these utilities will work right when put together. Even allowing for Murphy to create a few unexpected problems, assembling these components will be a feasible task. (The kernel will require closer communication and will be worked on by a small, tight group.)

If I get donations of money, I may be able to hire a few people full or part time. The salary won't be high by programmers' standards, but I'm looking for people for whom building community spirit is as important as making money. I view this as a way of enabling dedicated people to devote their full energies to working on GNU by sparing them the need to make a living in another way.

Why All Computer Users Will Benefit

Once GNU is written, everyone will be able to obtain good system software free, just like air.

This means much more than just saving everyone the price of a Unix license. It means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.

Complete system sources will be available to everyone. As a result, a user who needs changes in the system will always be free to make them himself, or hire any available programmer or company to make them for him. Users will no longer be at the mercy of one programmer or company which owns the sources and is in sole position to make changes.

Schools will be able to provide a much more educational environment by encouraging all students to study and improve the system code. Harvard's computer lab used to have the policy that no program could be installed on the system if its sources were not on public display, and upheld it by actually refusing to install certain programs. I was very much inspired by this.

Finally, the overhead of considering who owns the system software and what one is or is not entitled to do with it will be lifted.

Arrangements to make people pay for using a program, including licensing of copies, always incur a tremendous cost to society through the cumbersome mechanisms necessary to figure out how much (that is, which programs) a person must pay for. And only a police state can force everyone to obey them. Consider a space station where air must be manufactured at great cost: charging each breather per liter of air may be fair, but wearing the metered gas mask all day and all night is intolerable even if everyone can afford to pay the air bill. And the TV cameras everywhere to see if you ever take the mask off are outrageous. It's better to support the air plant with a head tax and chuck the masks.

Copying all or parts of a program is as natural to a programmer as breathing, and as productive. It ought to be as free.

Some Easily Rebutted Objections to GNU's Goals

> "Nobody will use it if it is free, because that means they can't rely on any support."
>
> "You have to charge for the program to pay for providing the support."

If people would rather pay for GNU plus service than get GNU free without service, a company to provide just service to people who have obtained GNU free ought to be profitable.

We must distinguish between support in the form of real programming work and mere hand-holding. The former is something one cannot rely on from a software vendor. If your problem is not shared by enough people, the vendor will tell you to get lost.

If your business needs to be able to rely on support, the only way is to have all the necessary sources and tools. Then you can hire any available person to fix your problem; you are not at the mercy of any individual. With Unix, the price of sources puts this out of consideration for most businesses. With GNU this will be easy. It is still possible for there to be no available competent person, but this problem cannot be blamed on distibution arrangements. GNU does not eliminate all the world's problems, only some of them.

Meanwhile, the users who know nothing about computers need handholding: doing things for them which they could easily do themselves but don't know how.

Such services could be provided by companies that sell just hand-holding and repair service. If it is true that users would rather spend money and get a product with service, they will also be willing to buy the service having got the product free. The service companies will compete in quality and price; users will not be tied to any particular one. Meanwhile, those of us who don't need the service should be able to use the program without paying for the service.

> "You cannot reach many people without advertising, and you must charge for the program to support that."
>
> "It's no use advertising a program people can get free."

There are various forms of free or very cheap publicity that can be used to inform numbers of computer users about something like GNU. But it may be true that one can reach more microcomputer users with advertising. If this is really so, a business which advertises the service of copying and mailing GNU for a fee ought to be successful enough to pay for its advertising and more. This way, only the users who benefit from the advertising pay for it.

On the other hand, if many people get GNU from their friends, and such companies don't succeed, this will show that advertising was not really necessary to spread GNU. Why is it that free market advocates don't want to let the free market decide this?

> "My company needs a proprietary operating system to get a competitive edge."

GNU will remove operating system software from the realm of competition. You will not be able to get an edge in this area, but neither will your competitors be able to get an edge over you. You and they will compete in other areas, while benefitting mutually in this one. If your business is selling an operating system, you will not like GNU, but that's tough on you. If your business is something else, GNU can save you from being pushed into the expensive business of selling operating systems.

I would like to see GNU development supported by gifts from many manufacturers and users, reducing the cost to each.

"Don't programmers deserve a reward for their creativity?"

If anything deserves a reward, it is social contribution. Creativity can be a social contribution, but only in so far as society is free to use the results. If programmers deserve to be rewarded for creating innovative programs, by the same token they deserve to be punished if they restrict the use of these programs.

"Shouldn't a programmer be able to ask for a reward for his creativity?"

There is nothing wrong with wanting pay for work, or seeking to maximize one's income, as long as one does not use means that are destructive. But the means customary in the field of software today are based on destruction.

Extracting money from users of a program by restricting their use of it is destructive because the restrictions reduce the amount and the ways that the program can be used. This reduces the amount of wealth that humanity derives from the program. When there is a deliberate choice to restrict, the harmful consequences are deliberate destruction.

The reason a good citizen does not use such destructive means to become wealthier is that, if everyone did so, we would all become poorer from the mutual destructiveness. This is Kantian ethics; or, the Golden Rule. Since I do not like the consequences that result if everyone hoards information, I am required to consider it wrong for one to do so. Specifically, the desire to be rewarded for one's creativity does not justify depriving the world in general of all or part of that creativity.

"Won't programmers starve?"

I could answer that nobody is forced to be a programmer. Most of us cannot manage to get any money for standing on the street and making faces. But we are not, as a result, condemned to spend our lives standing on the street making faces, and starving. We do something else.

But that is the wrong answer because it accepts the questioner's implicit assumption: that without ownership of software, programmers cannot possibly be paid a cent. Supposedly it is all or nothing.

The real reason programmers will not starve is that it will still be possible for them to get paid for programming; just not paid as much as now.

Restricting copying is not the only basis for business in software. It is the most common basis because it brings in the most money. If it were prohibited, or rejected by the customer, software business would move to other bases of organization which are now used less often. There are always numerous ways to organize any kind of business.

Probably programming will not be as lucrative on the new basis as it is now. But that is not an argument against the change. It is not considered an injustice that sales clerks make the salaries that they now do. If programmers made the same, that would not be an injustice either. (In practice they would still make considerably more than that.)

"Don't people have a right to control how their creativity is used?"

"Control over the use of one's ideas" really constitutes control over other people's lives; and it is usually used to make their lives more difficult.

People who have studied the issue of intellectual property rights carefully (such as lawyers) say that there is no intrinsic right to intellectual property. The kinds of supposed intellectual property rights that the government recognizes were created by specific acts of legislation for specific purposes.

For example, the patent system was established to encourage inventors to disclose the details of their inventions. Its purpose was to help society rather than to help inventors. At the time, the life span of 17 years for a patent was short compared with the rate of advance of the state of the art. Since patents are an issue only among manufacturers, for whom the cost and effort of a license agreement are small compared with setting up production, the patents often do not do much harm. They do not obstruct most individuals who use patented products.

The idea of copyright did not exist in ancient times, when authors frequently copied other authors at length in works of non-fiction. This practice was useful, and is the only way many authors's works have survived even in part. The copyright system was created expressly for the purpose of encouraging authorship. In the domain for which it was invented—books, which could be copied economically only on a printing press—it did little harm, and did not obstruct most of the individuals who read the books.

All intellectual property rights are just licenses granted by society because it was thought, rightly or wrongly, that society as a whole would benefit by granting them. But in any particular situation, we have to ask: are we really better off granting such license? What kind of act are we licensing a person to do?

The case of programs today is very different from that of books a hundred years ago. The fact that the easiest way to copy a program is from one neighbor to another, the fact that a program has both source code and object code which are distinct, and the fact that a program is used rather than read and enjoyed, combine to create a situation in which a person who enforces a copyright is harming society as a whole both materially and spiritually; in which a person should not do so regardless of whether the law enables him to.

"Competition makes things get done better."

The paradigm of competition is a race: by rewarding the winner, we encourage everyone to run faster. When capitalism really works this way, it does a good job; but its defenders are wrong in assuming it always works this way. If the runners forget why the reward is offered and become intent on winning, no matter how, they may find other strategies—such as, attacking other runners. If the runners get into a fist fight, they will all finish late.

Proprietary and secret software is the moral equivalent of runners in a fist fight. Sad to say, the only referree we've got does not seem to object to fights; he just regulates them ("For every ten yards you run, you can fire one shot"). He really ought to break them up, and penalize runners for even trying to fight.

"Won't everyone stop programming without a monetary incentive?"

Actually, many people will program with absolutely no monetary incentive. Programming has an irresistible fascination for some people, usually the people who are best at it. There is no shortage of professional musicians who keep at it even though they have no hope of making a living that way.

But really this question, though commonly asked, is not appropriate to the situation. Pay for programmers will not disappear, only become less. So the right question is, will anyone program with a reduced monetary incentive? My experience shows that they will.

For more than ten years, many of the world's best programmers worked at the Artificial Intelligence Lab for far less money than they could have had anywhere else. They got many kinds of non-monetary rewards: fame and appreciation, for example. And creativity is also fun, a reward in itself.

Then most of them left when offered a chance to do the same interesting work for a lot of money.

What the facts show is that people will program for reasons other than riches; but if given a chance to make a lot of money as well, they will come to expect and demand it. Low-paying organizations do poorly in competition with high-paying ones, but they do not have to do badly if the high-paying ones are banned.

> "We need the programmers desperately. If they demand that we stop helping our neighbors, we have to obey."

You're never so desperate that you have to obey this sort of demand. Remember: millions for defense, but not a cent for tribute!

> "Programmers need to make a living somehow."

In the short run, this is true. However, there are plenty of ways that programmers could make a living without selling the right to use a program. This way is customary now because it brings programmers and businessmen the most money, not because it is the only way to make a living. It is easy to find other ways if you want to find them. Here are a number of examples.

A manufacturer introducing a new computer will pay for the porting of operating systems onto the new hardware.

The sale of teaching, hand-holding and maintenance services could also employ programmers.

People with new ideas could distribute programs as freeware, asking for donations from satisfied users, or selling hand-holding services. I have met people who are already working this way successfully.

Users with related needs can form users' groups, and pay dues. A group would contract with programming companies to write programs that the group's members would like to use.

All sorts of development can be funded with a Software Tax:

> Suppose everyone who buys a computer has to pay x percent of the price as a software tax. The government gives this to an agency like the NSF to spend on software development.
>
> But if the computer buyer makes a donation to software development himself, he can take a credit against the tax. He can donate to the project of his own choosing—often, chosen because he hopes to use the results when it is done. He can take a credit for any amount of donation up to the total tax he had to pay.
>
> The total tax rate could be decided by a vote of the payers of the tax, weighted according to the amount they will be taxed on.
>
> The consequences:
>
> - The computer-using community supports software development.
> - This community decides what level of support is needed.
> - Users who care which projects their share is spent on can choose this for themselves.

In the long run, making programs free is a step toward the post-scarcity world, where nobody will have to work very hard just to make a living. People will be free to devote themselves to activities that are fun, such as programming, after spending the necessary ten hours a week on required tasks such as legislation, family counseling, robot repair and asteroid prospecting. There will be no need to be able to make a living from programming.

We have already greatly reduced the amount of work that the whole society must do for its actual productivity, but only a little of this has translated itself into leisure for workers because much nonproductive activity is required to accompany productive activity. The main causes of this are bureaucracy and isometric struggles against competition. Free software will greatly reduce these drains in the area of software production. We must do this, in order for technical gains in productivity to translate into less work for us.

Glossary

Abbrev An abbrev is a text string which expands into a different text string when present in the buffer. For example, you might define a short word as an abbrev for a long phrase that you want to insert frequently. See chapter 25 [Abbrevs], page 139.

Aborting Aborting means getting out of a recursive edit (q.v.)). The commands `C-≠]` and `M-x top-level` are used for this. See section 30.1 [Quitting], page 173.

Auto Fill mode
Auto Fill mode is a minor mode in which text that you insert is automatically broken into lines of fixed width. See section 22.6 [Filling], page 106.

Balance Parentheses
Emacs can balance parentheses manually or automatically. Manual balancing is done by the commands to move over balanced expressions (see section 23.2 [Lists], page 112). Automatic balancing is done by blinking the parenthesis that matches one just inserted (see section 23.5 [Matching Parens], page 120).

Bind To bind a key is to change its binding (q.v.).

Binding A key gets its meaning in Emacs by having a *binding* which is a command (q.v.), a Lisp function that is run when the key is typed. Customization often involves rebinding a character to a different command function. The bindings of all keys are recorded in the keymaps (q.v.). See section 2.3 [Commands], page 12.

Blank Lines
Blank lines are lines that contain only whitespace. Emacs has several commands for operating on the blank lines in the buffer.

Buffer The buffer is the basic editing unit; one buffer corresponds to one piece of text being edited. You can have several buffers, but at any time you are editing only one, the 'selected' buffer, though several can be visible when you are using multiple windows. See chapter 18 [Buffers], page 85.

Buffer Selection History
Emacs keeps a buffer selection history which records how recently each Emacs buffer has been selected. This is used for choosing a buffer to select. See chapter 18 [Buffers], page 85.

C- '`C`' in the name of a character is an abbreviation for Control. See section 2.1 [Characters], page 11.

C-M- '`C-M-`' in the name of a character is an abbreviation for Control-Meta. See section 2.1 [Characters], page 11.

Case Conversion
Case conversion means changing text from upper case to lower case or vice versa. See section 22.7 [Case], page 108, for the commands for case conversion.

Characters

Characters form the contents of an Emacs buffer; also, Emacs commands are invoked by keys (q.v.), which are sequences of one or more characters. See section 2.1 [Characters], page 11.

Command

A command is a Lisp function specially defined to be able to serve as an key binding in Emacs. When you type a key (q.v.), its binding (q.v.) is looked up in the relevant keymaps (q.v.) to find the command to run. See section 2.3 [Commands], page 12.

Command Name

A command name is the name of a Lisp symbol which is a command (see section 2.3 [Commands], page 12). You can invoke any command by its name using `M-x` (see chapter 7 [M-x], page 31).

Comments

A comment is text in a program which is intended only for humans reading the program, and is marked specially so that it will be ignored when the program is loaded or compiled. Emacs offers special commands for creating, aligning and killing comments. See section 23.6 [Comments], page 120.

Compilation

Compilation is the process of creating an executable program from source code. Emacs has commands for compiling files of Emacs Lisp code (see section 24.3 [Lisp Libraries], page 132) and programs in C and other languages (see section 24.1 [Compilation], page 131).

Completion

Completion is what Emacs does when it automatically fills out an abbreviation for a name into the entire name. Completion is done for minibuffer (q.v.) arguments, when the set of possible valid inputs is known; for example, on extended command names, buffer names, and file names. Completion occurs when `TAB`, `SPC` or `RET` is typed. See section 6.3 [Completion], page 27.

Continuation Line

When a line of text is longer than the width of the screen, it is takes up more than one screen line when displayed. We say that the text line is continued, and all screen lines used for it after the first are called continuation lines. See chapter 4 [Basic Editing], page 17.

Control-Character

ASCII characters with octal codes 0 through 040, and also code 0177, do not have graphic images assigned to them. These are the control characters. Any control character can be typed by holding down the `CTRL` key and typing some other character; some have special keys on the keyboard. `RET`, `TAB`, `ESC`, `LFD` and `DEL` are all control characters. See section 2.1 [Characters], page 11.

Current Buffer
The current buffer in Emacs is the Emacs buffer on which most editing commands operate. You can select any Emacs buffer as the current one. See chapter 18 [Buffers], page 85.

Current Line
The line point is on (see section 1.1 [Point], page 5).

Current Paragraph
The paragraph that point is in. If point is between paragraphs, the current paragraph is the one that follows point. See section 22.4 [Paragraphs], page 104.

Current Defun
The defun (q.v.) that point is in. If point is between defuns, the current defun is the one that follows point. See section 23.3 [Defuns], page 114.

Cursor The cursor is the rectangle on the screen which indicates the position called point (q.v.) at which insertion and deletion takes place. Often people speak of 'the cursor' v hen, strictly speaking, they mean 'point'. See chapter 4 [Basic Editing], page 17.

Customization
Customization is making minor changes in the way Emacs works. It is often done by setting variables (see section 29.2 [Variables], page 159) or by rebinding keys (see section 29.4 [Keymaps], page 166).

Default Argument
The default for an argument is the value that will be assumed if you do not specify one. When the minibuffer is used to read an argument, the default argument is used if you just type `RET`. See chapter 6 [Minibuffer], page 25.

Default File Name
When a file name arguments is specified, any file name components that you do not specify are taken from the corresponding components of the default file name. In Emacs, the default file name is normally the name of the file visited in the current buffer.

Defun A defun is a list at the top level of parenthesis or bracket structure in a program. It is so named because most such lists in Lisp programs are calls to the Lisp function `defun`. See section 23.3 [Defuns], page 114.

`DEL` `DEL` is a character used as a command to delete one character of text. See chapter 4 [Basic Editing], page 17.

Deletion Deletion means erasing text without saving it. Emacs deletes text only when it is expected not to be worth saving (all whitespace, or only one character). The alternative is killing (q.v.). See section 10.1 [Killing], page 41.

Deletion of Files
Deletion of a file means erasing it from the file system. See section 17.3 [Other File Commands], page 83.

Directory
: Files in the Unix file system are grouped into file directories. See section 16.7 [Directories], page 79.

Dired
: Dired is the Emacs facility that displays the contents of a file directory and allows you to "edit the directory", performing operations on the files in the directory. See chapter 17 [Dired], page 81.

Disabled Command
: A disabled command is one that you may not run without special confirmation. The usual reason for disabling a command is that it is confusing for beginning users. See section 29.4.4 [Disabling], page 169.

Dot
: An alternate name for 'point' (q.v.), often used in Emacs command names and on-line documentation. See section 1.1 [Point], page 5.

Echo Area
: The echo area is the bottom line of the screen, used for echoing the arguments to commands, for asking questions, and printing brief messages (including error messages). See section 1.2 [Echo Area], page 6.

Echoing
: Echoing is acknowledging the receipt of commands by displaying them (in the echo area). Emacs never echoes single-character keys; longer commands echo only if you pause while typing them.

Error Messages
: Error messages are single lines of output printed by Emacs when the user asks for something impossible to do (such as, killing text forward when point is at the end of the buffer). They appear in the echo area, accompanied by a beep.

`ESC`
: `ESC` is a character, used to end incremental searches and as a prefix for typing Meta characters on keyboards lacking a `META` key.

Fill Prefix
: The fill prefix is a string that should be expected at the beginning of each line when filling is done. It is not regarded as part of the text to be filled. See section 22.6 [Filling], page 106.

Filling
: Filling text means moving text from line to line so that all the lines are approximately the same length. See section 22.6 [Filling], page 106.

Global
: Global means 'independent of the current environment; in effect throughout Emacs'. It is the opposite of local (q.v.). Particular examples of the use of 'global' appear below.

Global Abbrev
: A global definition of an abbrev (q.v.) is effective in all major modes that do not have local (q.v.) definitions for the same abbrev. See chapter 25 [Abbrevs], page 139.

Global Keymap
: The global keymap (q.v.) contains key bindings that are in effect except when over-

riden by local key bindings in a major mode's local keymap (q.v.). See section 29.4 [Keymaps], page 166.

Global Substitution
Global substitution means replacing one string by another string through a large amount of text. See section 14.7 [Replace], page 63.

Global Variable
The global value of a variable (q.v.) takes effect in all buffers that do not have their own local (q.v.) values for the variable. See section 29.2 [Variables], page 159.

Graphic Character
Graphic characters are those assigned pictorial images rather than just names. All the non-Meta (q.v.) characters except for the Control (q.v.) characters are graphic characters. These include letters, digits, punctuation, and spaces; they do not include `RET` or `ESC`. In Emacs, typing a graphic character inserts that character. See chapter 4 [Basic Editing], page 17.

Grinding Grinding means reformatting a program so that it is indented according to its structure. See chapter 21 [Indentation], page 97.

Hardcopy
Hardcopy means printed output. Emacs has commands for making printed listings of text in Emacs buffers. See section 28.2 [Hardcopy], page 155.

`HELP` You can type `HELP` at any time to ask what options you have, or to ask what any command does. `HELP` is really `Control-h`. See chapter 8 [Help], page 33.

Indentation
Indentation means blank space at the beginning of a line. Most programming languages have conventions for using indentation to illuminate the structure of the program, and Emacs has special features to help you set up the correct indentation. See chapter 21 [Indentation], page 97.

Insertion Insertion means copying text into the buffer, either from the keyboard or from some other place in Emacs.

Justification
Justification means adding extra spaces to lines of text to make them come exactly to a specified width. See section 22.6 [Filling], page 106.

Keyboard Macros
Keyboard macros are a way of defining new Emacs commands from sequences of existing ones, with no need to write a Lisp program. See section 29.3 [Keyboard Macros], page 164.

Key A key is a character or sequence of characters which, when typed by the user, fully specifies one action to be performed by Emacs. For example, `X` and `Control-f` and `Control-x m` are keys. Keys derive their meanings from being bound (q.v.) to commands (q.v.). See section 2.2 [Keys], page 12.

Keymap The keymap is the data structure that records the bindings (q.v.) of keys to the commands that they run. For example, the keymap binds the character `C-n` to the command function **next-line**. See section 29.4 [Keymaps], page 166.

Kill Ring
The kill ring is where all text you have killed recently is saved. You can reinsert any of the killed text still in the ring; this is called yanking (q.v.). See section 10.2 [Yanking], page 43.

Killing Killing means erasing text and saving it on the kill ring so it can be yanked (q.v.) later. Most Emacs commands to erase text do killing, as opposed to deletion (q.v.). See section 10.1 [Killing], page 41.

List A list is, approximately, a text string beginning with an open parenthesis and ending with the matching close parenthesis. In C mode and other non-Lisp mode groupings surrounded by other kinds of matched delimiters appropriate to the language, such as braces, are also considered lists. Emacs has special commands for many operations on li ts. See section 23.2 [Lists], page 112.

Local Local means 'in effect only in a particular context'; the relevant kind of context is a particular function execution, a particular buffer, or a particular major mode. It is the opposite of 'global' (q.v.). Specific uses of 'local' in Emacs terminology appear below.

Local Abbrev
A local abbrev definition is effective only if a particular major mode is selected. In that major mode, it overrides any global definition for the same abbrev. See chapter 25 [Abbrevs], page 139.

Local Keymap
A local keymap is used in a particular major mode; the key bindings (q.v.) in the current local keymap override global bindings of the same keys. See section 29.4 [Keymaps], page 166.

Local Variable
A local value of a variable (q.v.) applies to only one buffer. See section 29.2 [Variables], page 159.

M- `M-` in the name of a character is an abbreviation for `META`, one of the modifier keys that can accompany any character. See section 2.1 [Characters], page 11.

M-x `M-x` is the key which is used to call an Emacs command by name. This is how commands that are not bound to keys are called. See chapter 7 [M-x], page 31.

Mail Mail means messages sent from one user to another through the computer system. Emacs has commands for composing and sending mail, and for reading and editing the mail you have received. See section 27.2 [Mail], page 150.

Major Mode
The major modes are a mutually exclusive set of options each of which configures

Emacs for editing a certain sort of text. Ideally, each programming language has its own major mode. See chapter 20 [Major Modes], page 95.

Mark The mark points to a position in the text. It specifies one end of the region (q.v.), point being the other end. Many commands operate on all the text from point to the mark. See chapter 9 [Mark], page 37.

Mark Ring
The mark ring is used to hold several recent previous locations of the mark, just in case you want to move back to them. See section 9.3 [Mark Ring], page 39.

Message See 'mail'.

Meta Meta is the name of a modifier bit which a command character may have. It is present in a character if the character is typed with the `META` key held down. Such characters are given names that start with `Meta-`. For example, `Meta-<` is typed by holding down `META` and typing `<` (which itself is done by holding down `SHIFT` and typing `,`). See section 2.1 [Characters], page 11.

Meta Character
A Meta character is one whose character code includes the Meta bit.

Minibuffer
The minibuffer is the window that appears when necessary inside the echo area (q.v.), used for reading arguments to commands. See chapter 6 [Minibuffer], page 25.

Minor mode
A minor mode is an optional feature of Emacs which can be switched on or off independently of all other features. Each minor mode has a command to turn it on or off. See section 29.1 [Minor Modes], page 159.

Mode line
The mode line is the line at the bottom of each text window (q.v.), which gives status information on the buffer displayed in that window. See section 1.3 [Mode Line], page 7.

Modified Buffer
A buffer (q.v.) is modified if its text has been changed since the last time the buffer was saved (or when it was created, if it has never been saved). See section 16.3 [Saving], page 73.

Moving Text
Moving text means erasing it from one place and inserting it in another. This is done by killing (q.v.) and then yanking (q.v.). See section 10.1 [Killing], page 41.

Named Mark
A named mark is a register (q.v.) in its role of recording a location in text so that you can move point to that location. See chapter 11 [Registers], page 49.

Narrowing
Narrowing means creating a restriction (q.v.) that limits editing in the current buffer

to only a part of the text in the buffer. Text outside that part is inaccessible to the user until the boundaries are widened again, but it is still there, and saving the file saves it all. See chapter 28 [Narrowing], page 153.

Newline `LFD` characters in the buffer terminate lines of text and are called newlines. See section 2.1 [Characters], page 11.

Numeric Argument
A numeric argument is a number, specified before a command, to change the effect of the command. Often the numeric argument serves as a repeat count. See chapter 5 [Arguments], page 23.

Option An option is a variable (q.v.) that exists so that you can customize Emacs by giving it a new value. See section 29.2 [Variables], page 159.

Overwrite Mode
Overwrite mode is a minor mode. When it is enabled, ordinary text characters replace the existing text after point rather than pushing it to the right. See section 29.1 [Minor Modes], page 159.

Page A page is a unit of text, delimited by formfeed characters (ASCII ^L, code 014) coming at the beginning of a line. Some Emacs commands are provided for moving over and operating on pages. See section 22.5 [Pages], page 105.

Paragraphs
Paragraphs are the medium-size unit of English text. There are special Emacs commands for moving over and operating on paragraphs. See section 22.4 [Paragraphs], page 104.

Parsing We say that Emacs parses words or expressions in the text being edited. Really, all it knows how to do is find the other end of a word or expression. See section 29.5 [Syntax], page 169.

Point Point is the place in the buffer at which insertion and deletion occur. Point is considered to be between two characters, not at one character. The terminal's cursor (q.v.) indicates the location of point. See chapter 4 [Basic], page 17.

Prefix Key
A prefix key is a key (q.v.) whose sole function is to introduce a set of multi-character keys. `Control-x` is an example of prefix key; thus, any sequence of `C-x` followed by one other character is also a legitimate key. See section 2.2 [Keys], page 12.

Prompt A prompt is text printed to ask the user for input. Printing a prompt is called *prompting*. Emacs prompts always appear in the echo area (q.v.). One kind of prompting happens when the minibuffer is used to read an argument (see chapter 6 [Minibuffer], page 25); the echoing which happens when you pause in the middle of typing a multi-character key is also a kind of prompting (see section 1.2 [Echo Area], page 6).

Quitting Quitting means cancelling a partially typed command or a running command, using `C-g`. See section 30.1 [Quitting], page 173.

Quoting Quoting means depriving a character of its usual special significance. It is usually done with `Control-q`. What constitutes special significance depends on the context and on convention. For example, an "ordinary" character as an Emacs command inserts itself; so in this context, a special character is any character that does not normally insert itself (such as DEL, for example), and quoting it makes it insert itself as if it were not special. Not all contexts allow quoting. See chapter 4 [Basic Editing], page 17.

Read-only Buffer
A read-only buffer is one whose text you are not allowed to change. Normally Emacs makes buffers read-only when they contain text which has a special significance to Emacs; for example, Dired buffers. Visiting a file that is write protected also makes a read-only buffer. See chapter 18 [Buffers], page 85.

Recursive Editing Level
A recursive editing level is a state in which part of the execution of a command involves asking the user to edit some text. This text may or may not be the same as the text to which the command was applied. The mode line indicates recursive editing levels with square brackets ('`[`' and '`]`'). See section 27.1 [Recursive Edit], page 149.

Redisplay
Redisplay is the process of correcting the image on the screen to correspond to changes that have been made in the text being edited. See chapter 1 [Screen], page 5.

Region The region is the text between point (q.v.) and the mark (q.v.). Many commands operate on the text of the region. See chapter 9 [Mark], page 37.

Registers Registers are named slots in which text or buffer positions or rectangles can be saved for later use. See chapter 11 [Registers], page 49.

Replacement
See 'global substitution'.

Restriction
A restriction in a buffer makes some of the text at the beginning or end of the buffer, or both, temporarily invisible and inaccessible. Creating a restriction on a buffer is called narrowing, and removing one is called widening. See chapter 28 [Narrowing], page 153.

RET RET is an Emacs command to insert a newline into the text. It is also used to terminate most arguments read in the minibuffer (q.v.). See section 2.1 [Characters], page 11.

Saving Saving a buffer means copying its text into the file that was visited (q.v.) in that buffer. This is the way text in files actually gets changed by your Emacs editing. See section 16.3 [Saving], page 73.

Scrolling Scrolling means shifting the text in the Emacs window so as to see a different part of the buffer. See chapter 13 [Display], page 53.

Searching

Searching means moving point to the next occurrence of a specified string. See chapter 14 [Search], page 55.

Selecting Selecting a buffer means making it the current (q.v.) buffer. See chapter 18 [Buffers], page 85.

Self-documentation
Self-documentation is the feature of Emacs which can tell you what any command does, or give you a list of all commands related to a topic you specify. You ask for self-documentation with the HELP character. See chapter 8 [Help], page 33.

Sentences
Emacs has commands for moving by or killing by sentences. See section 22.3 [Sentences], page 103.

Sexp A sexp (short for 's-expression') is the basic syntactic unit of Lisp in its textual form: either a list, or Lisp atom. Many Emacs commands operate on sexps. The term 'sexp' is generalized to languages other than Lisp, to mean a syntactically recognizable expression. See section 23.2 [Lists], page 112.

Simultaneous Editing
Simultaneous editing means two users modifying the same file at once. Simultaneous editing if not detected can cause one user to lose his work. Emacs detects all cases of simultaneous editing and warns the user to investigate them. See section 16.4.1 [Simultaneous Editing], page 76.

String Substitution
See 'global substitution'.

Syntax Table
The syntax table tells Emacs which characters are part of a word, which characters balance each other like parentheses, etc. See section 29.5 [Syntax], page 169.

Tag Table
A tag table is a file that serves as an index to the function definitions in one or more other files. See section 23.10 [Tags], page 125.

Text Two meanings (see chapter 22 [Text], page 101):

- Data consisting of a sequence of characters. The contents of an Emacs buffer are always text in this sense.
- Data consisting of written human language, as opposed to programs, or following the stylistic conventions of human language.

Top Level
Top level is the normal state of Emacs, in which you are editing the text of the file you have visited. You are at top level whenever you are not in a recursive editing level (q.v.) or the minibuffer (q.v.), and not in the middle of a command. You can get back to top level by aborting (q.v.) and quitting (q.v.). See section 30.1 [Quitting], page 173.

Transposition
Transposing two units of text means putting each one into the place formerly occupied by the other. There are Emacs commands to transpose two adjacent characters, words, sexps (q.v.) or lines (see section 15.2 [Transposition], page 68).

Truncation
Truncating text lines in the display means leaving out any text on a line that does not fit within the right margin of the window displaying it. See also 'continuation line'. See chapter 4 [Basic Editing], page 17.

Undoing Undoing means making your previous editing go in reverse, bringing back the text that existed earlier in the editing session. See chapter 12 [Undo], page 51.

Variable A variable is an object in Lisp that can store an arbitrary value. Emacs uses some variables for internal purposes, and has others (known as 'options' (q.v.)) just so that you can set their values to control the behavior of Emacs. The variables used in Emacs that you are likely to be interested in are listed in the Variables Index in this manual. See section 29.2 [Variables], page 159, for information on variables.

Visiting Visiting a file means loading its contents into a buffer (q.v.) where they can be edited. See section 16.2 [Visiting], page 72.

Wall Chart
The wall chart is a very brief Emacs reference sheet giving one line of information about each short command. A copy of the wall chart appears in this manual.

Whitespace
Whitespace is any run of consecutive formatting characters (space, tab, newline, and backspace).

Widening
Widening is removing any restriction (q.v.) on the current buffer; it is the opposite of narrowing (q.v.). See chapter 28 [Narrowing], page 153.

Window Emacs divides the screen into one or more windows, each of which can display the contents of one buffer (q.v.) at any time. See chapter 1 [Screen], page 5, for basic information on how Emacs uses the screen. See chapter 19 [Windows], page 91, for commands to control the use of windows.

Word Abbrev
Synonymous with 'abbrev'.

Word Search
Word search is searching for a sequence of words, considering the punctuation between them as insignificant. See section 14.3 [Word Search], page 58.

Yanking Yanking means reinserting text previously killed. It can be used to undo a mistaken kill, or for copying or moving text. See section 10.2 [Yanking], page 43.

Key (Character) Index

Command Index

Variable Index

Concept Index

CONNOCHAETES
TAURINUS