JOVE User Manual
Version 4.17

Jonathan Payne
(with revisions by Doug Kingston, Mark Seiden, D. Hugh Redelmeieir, Mark Moraes and Charles Lindsey)

1. Introduction

JOVE* is an advanced, self-documenting, customizable, display editor. It (and this tutorial introduction) are based on the original EMACS editor and user manual written at M.I.T. by Richard Stallman†.

JOVE is considered a display editor because normally the text being edited is visible on the screen and is updated automatically as you type your commands. What You See Is What You Get.

JOVE provides many facilities that go beyond simple insertion and deletion. Some of the more advanced features are:

cut and paste (or kill and yank in our terminology);
search and replace using regular-expressions;
multiple files, buffers and windows available simultaneously;
filling of text, both on demand and as you type;
manipulation of words, lines, sentences and paragraphs;
automatic indentation of programs;
automatic location of procedure definitions;
executing programs, capturing their output in buffers;
automatic location of spelling and compilation errors;
parenthesis matching.

JOVE is self-documenting insofar as you can call up descriptions of commands, variables and key bindings.

JOVE is customizable insofar as you can

change its behavior by changing appropriate variables;
change its behavior by setting appropriate modes;
automatically set the modes for a buffer from its filename;
define macros to perform complex tasks;
change the key bindings to match features of the particular keyboard.

All of these options can be exercised by the system administrator, or by the user at startup, or even in the middle of a job.

Finally, mouse support is available (on screens which support the X-Windows package from M.I.T.) using the front end program *xjove* (or alternatively via the mouse-reporting facilities of the terminal emulator *xterm*).

*JOVE stands for Jonathan's Own Version of EMACS.

†Although JOVE is meant to be compatible with EMACS, and indeed many of the basic commands are very similar, there are some major differences between the two editors, and you should not rely on their behaving identically.

2. The Organization of the Screen

JOVE divides the screen into several sections.

```
#include <stdio.h>

void main()
{
   printf("Hello world!\n");
   return 0;                                             < the Window
}

JOVE (C OvrWt)  [Main:1]  "hello.c"  *  /home/foo        < the Mode Line
: write-file (default hello.c) aloha.c□                  < the Message Line
```

2.1. The Window

The Window section is used to display the text you are editing. The terminal's cursor shows the position of *point*, the location at which editing takes place. While the cursor appears to point *at* a character, point should be thought of as between characters; it points *before* the character that the cursor appears to be on top of. Terminals have only one cursor, and when output is in progress it must appear where the typing is being done. This doesn't mean that point is moving; it is only that JOVE has no way of showing you the location of point except when the terminal is idle. In the example, the user is in the middle of issuing a *write-file* command, so the cursor is at the end of the message line.

2.1.1. Typeout

The lines of the window are usually available for displaying text but sometimes are pre-empted by typeout from certain commands (such as a listing of all the buffers). You can always recognize such *typeout* because it is terminated by either an **--end--** line or a **--more--** line. Most of the time, output from commands like these is only desired for a short period of time, usually just long enough to glance at it. When you have finished looking at the output, you can type Space to make the text reappear (usually a Space that you type inserts itself, but when there is typeout in the window, it does nothing but get rid of that). Any other command executes normally, *after* redrawing your text.

You will see **--more--** on the line above the last mode line when typeout from a command is too long to fit on the screen. It means that if you type a Space the next screenful of typeout will be printed. If you are not interested, typing ˆG will cause the rest of the output to be discarded. Typing any other key will discard the rest of the output and that key will be taken as the next keyboard input. Similarly, **--end--** signifies that typeout is complete; the same responses are accepted.

Sometimes you may wish to keep a permanent record of the typeout from these commands. To do this, set the variable *send-typeout-to-buffer* to *on*. The typeout will then be put into a newly-created buffer, which you can arrange to save to a permanent file.

2.2. The Mode Line

The Mode Line gives information about the window above it. There is a variable *mode-line* which determines the layout of the mode line. For the example above, this was set as described in the section on customizing JOVE.

(C OvrWt) shows that **C** is the name of the current *major mode* and that the Over Write *minor mode* is turned on.

At any time, JOVE can be in only one major mode. Currently there are four major modes: *Fundamental*, *Text*, *Lisp* and *C*. New ones may be added in the future.

The words which indicate which minor modes are turned on are:

> **Abbrev** meaning that *Word Abbrev* mode is on;
> **AI** meaning that *Auto Indent* mode is on;
> **Fill** meaning that *Auto Fill* mode is on;
> **OvrWt** meaning that *Over Write* mode is on;
> **RO** meaning that *Read Only* mode is on.
> **DBX** meaning that *DBX* mode is on.
> **Def** meaning that you are in the process of defining a keyboard macro.
> This is not really a mode, but it's useful to be reminded about it.

The meanings of these modes are described later in this document.

[Main:1] shows that the name of the currently selected *buffer* is **Main** and its number is **1**. Each buffer has its own name and holds a file being edited, which is how JOVE can hold several files at once. But at any given time you are editing only one of them, the *selected* buffer. When we speak of what some command does to "the buffer", we are talking about the currently selected buffer. Multiple buffers make it easy to switch around between several files, and then it is very useful that the mode line tells you which one you are editing at any time.

"hello.c" shows the name of the file being edited in buffer **Main**. This is also the default filename for commands that expect a filename as input, as can be seen in the message line which follows.

The ∗ in the mode line means that there are changes in the buffer that have not been saved in the file. If the buffer had not been changed since it was read in or last saved, there would be a minus instead.

Sometimes a file is changed ‘‘behind JOVE's back'': something changes the file (not the buffer) after it has been loaded into or saved from a buffer. This can be quite dangerous, so JOVE tests for this when it reads, writes, or finds the file. JOVE indicates the problem by displaying a # before the change indicator. It also asks for confirmation before performing the read or write.

/home/foo shows the name of the current directory.

15.23 shows the time.

2.3. The Message Line

The Message Line is reserved for printing messages and for accepting input from the user, such as filenames or search strings. When JOVE prompts for input, the cursor will temporarily appear on the bottom line, waiting for you to type a string. When you have finished typing your input, you can type a Return to send it to JOVE. If you change your mind about running the command that is waiting for input, you can type ˆG to abort, and you can then continue with your editing.

The message line and the list of filenames from the shell command that invoked JOVE are kept in a special buffer called *Minibuf* that can be edited like any other buffer. It is instructive to view the Minibuf in a window and to observe how it changes as parameters to commands are typed, and as the ˆN and ˆP functions are invoked.

2.4. Multiple Windows

The window area, described above, can in fact be split into several *windows*, each showing a different *buffer*, or possibly different parts of the same buffer. Each window has its own mode line beneath it. The methods of creating and destroying windows will be described presently.

3. Input Conventions

3.1. Keyboard Usage

In this manual, "Control" characters (that is, characters that are typed with the Control key and some other key at the same time) are represented by a circumflex (ˆ) followed by another character. Thus, ˆA is the character you get when you type A with the Control key (sometimes labeled CTRL) held down. Control characters in the JOVE buffer are displayed with a caret; thus, ˆA for Control-A. DEL is displayed as ˆ?, ESC as ˆ[.

If the keyboard has extra keys, such as Function keys, Arrow keys and the like, then JOVE can be customized to use them.

3.2. The Character Set

JOVE normally accepts the ASCII character set, with its 95 printing characters, including Space, (which appear on the screen as themselves) and its 33 Control characters (which, except for TAB, appear on the screen as, e.g. "ˆC"). There are, however, two characters that may not appear. One is the NL character (because it is always converted into a *line-separator*, which is not quite a character) and the other is the NUL character (ˆ@) which is used internally within JOVE to delimit lines (lines also have a maximum length, which is 1023 in most systems).

However, JOVE is "8-bit clean", so if your keyboard is able to produce all 256 8-bit characters, the extra ones will appear in octal (e.g. "\277"). Moreover, if your system supports the *Locale* facility (as most modern ones do), you may set the variable *lc-ctype* to "C" (the default, which corresponds to pure ASCII), or to "iso_8859_1" (which corresponds to the Latin-1 alphabet with a total of 192 printing characters, all of which JOVE should be able to display), or to any other *Locale* available on your system. The initial value of *lc-ctype* is taken from your LC_CTYPE environment variable, and otherwise defaults to "C". With each *Locale* JOVE will know which of the extra characters are upper-case letters, lower-case letters, etc.

3.3. Name Completion

JOVE knows the names of all sorts of objects, such as JOVE Commands, JOVE Variables, Macros, Keymaps, Buffers and even (with some help from the directories) Files. Since names must be entered often, JOVE has features to make this easier.

For many names, JOVE is willing to supply a default if you enter an empty answer. For example, when you are telling *select-buffer* which buffer to select, it will default to the previous buffer. When the prompt mentions a default, this is the value that will be used in place of an empty answer.

If the default isn't the name you want, name completion can help you enter a name. When you are prompted for a name, you need type only enough letters to make it unambiguous. At any point in the course of typing the name, you can type question mark (?) to see a list of all the relevant names which begin with the characters you have already typed; you can type Tab to have JOVE supply as many characters as it can; or you can type Return to terminate your input, or you can type Space to do both (supply the characters and terminate). For example, you are typing a Command and you have so far typed the letters "*au*" and you then type a question mark, you will see the list

```
auto-execute-command
auto-execute-macro
auto-fill-mode
auto-indent-mode
```

If you type a Return at this point, JOVE will complain by ringing the bell, because the letters you have typed do not unambiguously specify a single command. But if you type Tab or Space, JOVE will supply the characters "*to-*" because all commands that begin "*au*" also begin "*auto-*". You could then type the letter "f" followed by either Space or Return, and JOVE would complete and obey the entire command.

There are in fact two cases that can arise.

1. **The name you are typing is supposed to exist already** (Commands, Variables and Keymaps always, Macros and Buffers except when you are trying to create a new one).
   If you type Return and what you have typed is not an unambiguous prefix of any name of the right kind, you will hear the bell; otherwise, it will complete what you have typed and then use it. Tab will complete what it can (you can then type Return if it looks right). Space will complete what it can and use it if it then matches.
2. **The name you are typing may be a new one** (Files always, Macros (including the Keyboard Macro) and Buffers if you are allowed to create or rename one at that point).
   If you type Return, and it does not match any name, then it will take exactly what you have typed as a new name. Tab and Space try to complete as before.

If you type ˆR, it will insert a name that might be useful. Even if this name is not the one you wish to enter, it is often convenient to edit this name into the desired one. The inserted name will be the default (if there is one), or the current value (if there is one). When JOVE is asking for a command or variable name, ˆR will insert the last one named.

Buffers, keymaps, and macros are also numbered (if you type "?" when first prompted, you will see the numbers as well as the possible names), and the number may be used in place of the name.

3.3.1. Filename Completion

Whenever JOVE is prompting you for a filename, say in the *find-file* command, things happen as just described and Return always accepts the name just as it is (because you might be wanting to create a new file with a name similar to that of an existing one). The variable *bad-filename-extensions* contains a list of words separated by spaces which are to be considered bad filename extensions; any filename with one of these extensions will not be counted in filename completion. The default includes ".o" so if you have jove.c and jove.o in the same directory, the filename completion will not complain of an ambiguity because it will ignore jove.o.

When JOVE is prompting for a *filename*, it has the following extra functions:

ˆN	Insert the next filename from the argument list in the Minibuf.
ˆP	Insert the previous filename from the argument list in the Minibuf.

4. Commands and Variables

4.1. Commands

JOVE uses *commands* which have long names such as *next-line*. Then*k eys* such as ˆN are connected to commands through the *command dispatch table*. When we say that ˆN moves the cursor down a line, we are glossing over a distinction which is unimportant for ordinary use, but essential for simple customization: it is the command *next-line* which knows how to move down a line, and ˆN moves down a line because it is connected to that command. The name for this connection is a *binding*; we say that the key ˆN *is bound to* the command *next-line* (or vice versa). JOVE has many bindings already *built-in*, but you (or your system administrator) may also add your own, e.g. to make full use of any Function Keys provided on your particular keyboard.

Thus there may be three ways to refer to a command — by its full name, or by its standard (built-in) binding, or by your customized binding. Throughout this manual, we shall always use the standard bindings, followed by the full name (in italics). The standard bindings are designed to work on any ASCII keyboard, and can always be used so long as you (or your system administrator) have not actually changed them. But they are hard to remember, so you may well prefer to use your own, particularly if you always use the same terminal. See the section on Customization for more details.

Some terminals and modems cannot accept characters flat out at a reasonable baud rate, and therefore require the use of a flow control protocol using the characters ˆS and ˆQ (see the variable *allow-ˆS-and-ˆQ*). These characters cannot, therefore, be typed by the user. It has therefore been arranged that whenever a standard binding requires ˆS (ˆQ) to be typed, a spare standard binding for that facility is also provided in which ˆ\ (ˆˆ) can be typed in its place.

Not all commands are bound to keys. To invoke a command that isn't bound to a key, you can type the sequence ESC X, which is bound to the command *execute-named-command*. You will then be able to type the name of whatever command you want to execute on the message line.

4.2. Prefix Characters

Because there are more command names than keys, JOVE allows a sequence of keystrokes to be bound to a command. Usually, the first character of the sequence will be one of the two *prefix characters* ˆX or ESC. When you type such a prefix character JOVE will wait for the next character before deciding what to do. If you wait more than a second or so, JOVE will print the prefix character on the message line as a reminder and leave the cursor down there while you type the rest of the sequence. Many JOVE commands are bound to a 2-stroke sequence starting with ˆX or ESC. How the next character is interpreted depends on which of them you typed. For example, if you type ESC followed by B you will run *backward-word*, but if you type ˆX followed by B you will run *select-buffer*.

4.3. Variables

Sometimes 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. JOVE contains variables which are there so that you can change them if you want to customize. The variable's value may be examined by some command, and changing that value makes the command behave differently. However, the facilities provided are pretty limited: you cannot invent new variables, or use them for other than their built-in purposes, and their values apply globally to all buffers irrespective of mode settings.

set	Sets the value of a variable.
print	Displays the current value of a variable.

To set or change the value of a variable, type ESC X *set* <variable-name> <value><return>. Values may be *on* of *off* (for Boolean variables) or numbers (numeric variables) or strings (string variables). To inspect the current value of a variable, type ESC X print <variable-name><return>.

4.4. Giving Numeric Arguments to JOVE Commands

Many JOVE commands can be given a *numeric argument*. Many commands interpret the argument as a repetition count (possibly negative). For example, giving an argument of ten to the ˆF command (*forward-character*) moves forward ten characters. With these commands, no argument is equivalent to an argument of 1.

Some commands use the value of the argument, or even just its presence or absence, in highly idiosyncratic ways. For example, the commands which change the minor modes (such as *auto-fill-mode*) toggle the mode if there is no argument, but turn the more off with a zero argument, and on with any other argument.

The fundamental way of specifying an argument is to use ESC followed by the digits of the argument, for example, ESC 123 ESC G to go to line 123. Negative arguments are allowed, although not all commands know what to do with them. Unless otherwise stated, ESC Minus ... is equivalent to ESC Minus 1 ... . Note that when giving arguments to *sourced* commands (described later under Customization) different rules apply.

Typing ˆU means “supply an argument of 4”. Two such ˆU’s supply sixteen. Thus, ˆU ˆU ˆF moves forward sixteen characters. This is a good way to move forward quickly, since it moves about 1/4 of a line on most terminals. Other useful combinations are: ˆU ˆU ˆN (move down a good fraction of the screen), and ˆU ˆU ˆO (make “a lot” of blank lines).

There are other, terminal-dependent, ways of specifying arguments. They have the same effect but may be easier to type. If your terminal has a numeric keypad which sends something recognizably different from the ordinary digits, it is possible to customize JOVE to allow use of the numeric keypad for specifying arguments.

4.5. Help

To get a list of keys and their associated commands, you type ESC X *describe-bindings* (warning: the list runs to many screenfuls; type Space to see the next one, or ˆG when you have seen enough). If you want to describe a single key, ˆX ? (*describe-key*) will work. A description of an individual command is available by using ESC ? (*describe-command*), and descriptions of variables by using ESC X *describe-variable*. If you can’t remember the name of the thing you want to know about, ESC X *apropos* will tell you if a command or variable has a given string in its name. For example, ESC X *apropos describe* will list the names of the four describe commands just mentioned.

5. Basic Editing Commands

5.1. Inserting Text

To insert printing characters into the text, just type them. All such printing characters you type are inserted into the text at the cursor (that is, at *point*), and 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 correct text you have just inserted, you can use DEL. 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 typing a printing character and then type DEL, they cancel out.

To end a line and start typing a new one, type Return. Return operates by inserting a *line-separator*, so if you type Return in the middle of a line, you break the line in two. Because a line-separator behaves like a single character, you can type DEL at the beginning of a line to delete the line-separator and join it with the preceding line. Note that the line separator is **not** a character (it is not the ASCII NL character, for example) so that you cannot include it in search or replace strings.

As a special case, if you type Return at the end of a line and there are two or more empty lines just below it, JOVE does not insert a line-separator but instead merely moves to the next (empty) line. This behavior is convenient when you want to add several lines of text in the middle of a buffer. You can use the ˆO (*newline-and-backup*) command to “open” several empty lines at once; then you can insert the new text, filling up these empty lines. The advantage is that JOVE does not have to redraw the bottom part of the screen for each Return you type, as it would ordinarily. That “redisplay” can be both slow and distracting.

If you add too many characters to one line, without breaking it with Return, the line will grow too long to display on one screen line. When this happens, JOVE puts an “!” at the extreme right margin, and doesn’t bother to display the rest of the line unless the cursor happens to be in it. The “!” is not part of your text; conversely, even though you can’t see the rest of your line, it is still there, and if you break the line, the “!” will go away.

Direct insertion works for printing characters and space, but other characters act as editing commands and do not insert themselves. If you need to insert a control character, ESC, or DEL, you must first *quote* it by typing the ˆQ command (*quoted-insert*) first, for example ˆQ ˆC to insert a genuine ˆC.

5.2. Moving the Cursor

To do more than insert characters, you have to know how to move the cursor. Here are the commands for doing that.

ˆA	*beginning-of-line*	Move to the beginning of the line.
ˆE	*end-of-line*	Move to the end of the line.
ˆF or →	*forward-character*	Move forward over one character.
ˆB or ←	*backward-character*	Move backward over one character.
ˆN or ↓	*next-line*	Move down one line, vertically. If you start in the middle of one line, you end in the middle of the next.
ˆP or ↑	*previous-line*	Move up one line, vertically.
ESC <	*beginning-of-file*	Move to the beginning of the entire buffer.
ESC >	*end-of-file*	Move to the end of the entire buffer.
ESC ,	*beginning-of-window*	Move to the beginning of the visible window.
ESC .	*end-of-window*	Move to the end of the visible window.
ˆZ	*scroll-up*	Move the lines in the window upwards. If this brings the cursor outside of the window, it is automatically relocated.
ESC Z	*scroll-down*	Move the lines in the window downwards.

Observe the use of the arrow keys (→, ←, ↓ and ↑) as alternatives for ˆF, ˆB, ˆN and ˆP. These should be available on just about any terminal. You (or your system administrator) may find it convenient to bind other Function Keys available on your keyboard to some of these commands, especially if those keys already have appropriate engravings on them. See the section on Customizing JOVE.

5.3. Deleting Text

DEL	*delete-previous-character*	Delete the character before the cursor.
ˆD	*delete-next-character*	Delete the character after the cursor.
ESC \	*delete-white-space*	Delete spaces and tabs around point.
ˆX ˆO	*delete-blank-lines*	Delete blank lines around the current line.

You already know about the DEL command which deletes the character *before* the cursor. Another command, ˆD, deletes the character *after* the cursor, the one the cursor is “on top of” or “underneath”, causing the rest of the text on the line to shift left. Line-separators act like normal characters when deleted, so if ˆD is typed at the end of a line, that line and the next line are joined together.

The other delete commands are those which delete only formatting characters: spaces, tabs, and line-separators. ESC \ (*delete-white-space*) deletes all the spaces and tab characters before and after point. ˆX ˆO (*delete-blank-lines*) deletes all blank lines after the current line, and if the current line is blank deletes all the blank lines preceding the current line as well (leaving one blank line, the current line).

5.4. Files — Saving Your Work

The commands above are sufficient for creating text in the JOVE buffer. The more advanced JOVE commands just make things easier. But to keep any text permanently you must put it into a *file*. Files are the objects which uses for storing data for a length of time. To tell JOVE to read text into a file, choose a filename, such as *foo.bar*, and type ˆX ˆF *foo.bar*<return> (*find-file*). This reads the file*foo.bar* so that its contents appear in a ne w buffer on the screen for editing. Alternatively, type ˆX ˆR *foo.bar*<return> (*read-file*) to have the file appear in an existing buffer. You can make changes, and then save the file by typing ˆX ˆS (*save-file*). This makes the changes permanent and actually changes the file *foo.bar*. Until then, the changes are only inside JOVE, and the file *foo.bar* is not really changed. If the file *foo.bar* does not exist, and you want to create it, read it as if it did exist. When you save your text with ˆX ˆS, the file will be created then.

5.5. Exiting and Pausing — Leaving JOVE

The command ˆX ˆC (*exit-jove*) will terminate the JOVE session and return to the shell. If there are modified but unsaved buffers, JOVE will ask you for confirmation, and you can abort the command, look at what buffers are modified but unsaved using ˆX ˆB (*list-buffers*), save the valuable ones, and then exit. If what you want to do, on the other hand, is to *preserve* the editing session but return to the shell temporarily you can (under most modern versions of issue the command ESC S (*pause-jove*), do your work within your shell, and then return to JOVE using the *fg* command to resume editing at the point where you paused. Alternatively, for this sort of situation, you might consider using an *interactive shell* (that is, a shell in a JOVE window) which lets you use the editor to issue your commands and manipulate their output, while never leaving the editor (the interactive shell feature is described later).

6. Kill and Yank (or Cut and Paste)

Any editor needs a facility for dealing with large blocks of text — deleting them or moving them to other places. The usual terminology speaks of “Cut” (to remove a block of text), “Paste” (to replace it somewhere else) and “Copy” (to copy it for subsequent pasting without removal from its original place). For historical reasons, editors based on EMACS use the terms Kill, Yank and Copy with essentially the same meanings, and we shall continue to do so in this manual. However, it may be sensible, if your keyboard has keys marked Cut, Paste and Copy, to bind appropriate Kill, Yank and Copy commands to them as part of your local customization.

6.1. The Mark and the Region

In general, a command that processes an arbitrary part of the buffer must know where to start and where to stop. In JOVE, such commands usually operate on the text between *point* and *the mark*. On most terminals, the position of the mark is indicated by underlining. This body of text is called *the region*. To specify a region, you set point at one end of it and mark at the other. It doesn’t matter which one comes earlier in the text.

ˆ@	*set-mark*	Set the mark where point is.
ˆX ˆX	*exchange-point-and-mark*	Interchange point and mark.
	pop-mark	Move to the previous mark in the ring.

The way to set the mark is with the ˆ@ command or (on some terminals) the ˆSpace command. They set the mark where point is. Then you can move point away, leaving the mark behind. When the mark is set, “[Point pushed]” is printed on the message line.

For example, if you wish to convert part of the buffer to all upper-case, you can use the *case-region-upper* command, which operates on the text in the region. You can first go to the beginning of the text to be capitalized, put the mark there, move to the other end, and then type ESC X *case-region-upper*. Or, you can set the mark at the end of the text, move to the beginning, and then type the same thing.

On terminals with the requisite capabilities, the marked character is underlined. Otherwise, you have to remember where it is (the usual method is to set the mark and then use it soon). Alternatively, you can see where the mark is with the command ˆX ˆX 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.

6.2. The Ring of Marks

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, JOVE remembers 16 previous locations of the mark. Most commands that set the mark push the old mark onto this stack. To return to a marked location, use ˆU ˆ@ (equivalent to *pop-mark*). This moves point to where the mark was, and restores the mark from the stack of former marks. So repeated use of this command moves point to all of the old marks on the stack, one by one. Since the stack is actually a ring, enough uses of ˆU ˆ@ bring point back to where it was originally.

Some commands whose primary purpose is to move point a great distance take advantage of the stack of marks to give you a way to undo the command. The best example is ESC < (*beginning-of-file*), which moves to the beginning of the buffer. If there are more than 22 lines between the beginning of the buffer and point, ESC < sets the mark first, so that you can use ˆU ˆ@ or ˆX ˆX to go back to where you were. You can change the number of lines from 22 since it is kept in the variable *mark-threshold*. By setting it to 0, you can make these commands always set the mark and by setting it to a very large number you can make them never set it. If a command decides to set the mark, it prints the message [Point pushed].

6.3. Killing and Moving Text

The way of moving text with JOVE is to *kill* (cut) it, and *yank* (paste) it back again later in one or more places. This is very safe because the last several pieces of killed text are all remembered, and it is versatile because the many commands for killing syntactic units can also be used for moving those units.

6.3.1. 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 you can copy it to other parts of the buffer (even to a different 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 ˆD and DEL, which delete only one character at a time, and those commands that delete only spaces or line-separators. Commands that can destroy significant amounts of nontrivial data generally kill. A command's name and description will use the words *kill* or *delete* to say which it does.

ˆD	*delete-next-character*	Delete next character.
DEL	*delete-previous-character*	Delete previous character.
ESC \	*delete-white-space*	Delete spaces and tabs around point.
ˆX ˆO	*delete-blank-lines*	Delete blank lines around the current line.
ˆK	*kill-to-end-of-line*	Kill rest of line or one or more lines.
ˆW	*kill-region*	Kill the region (from point to mark).
ESC D	*kill-next-word*	Kill word.
ESC DEL	*kill-previous-word*	Kill word backwards.
ESC K	*kill-to-end-of-sentence*	Kill to end of sentence.
ˆX DEL	*kill-to-beginning-of-sentence*	Kill to beginning of sentence.
ESC ˆK	*kill-s-expression*	Kill from point to the end of an s-expression.

6.3.2. Deletion

The various delete commands have already been described. Actually, ˆD and DEL aren't always *delete* commands; if you give an argument, they *kill* instead. This prevents you from losing a great deal of text by typing a large argument to a ˆD or DEL.

6.3.3. Killing (and Copying) the region, and Yanking it back again

The commonest kill command is ˆW (*kill-region*), which kills everything between point and the mark*. With this command, you can kill and save contiguous characters, if you first set the mark at one end of them and then go to the other end.

ˆW	*kill-region*	Kill everything between point and mark.
ESC W	*copy-region*	Save the region without killing.

*Often users switch this binding from ˆW to ˆX ˆK because it is too easy to hit ˆW accidentally.

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

ˆY	*yank*	Yank (re-insert) the last killed text.
ESC Y	*yank-pop*	Replace re-inserted killed text with the previously killed text.

Killed text is pushed onto a *ring buffer* called the *kill ring* that remembers the last sixteen blocks of text that were killed (why it is called a ring buffer will be explained below). The command ˆY (*yank*) reinserts the text of the most recent kill. The yanked text becomes the new region. Thus, a single ˆY undoes the ˆW and vice versa.

If you wish to copy a block of text, you might want to use ESC W (*copy-region*), which copies the region into the kill ring without removing it from the buffer.

There is only one kill ring shared among all the buffers. After reading a new file or selecting a new buffer, whatever was last killed in the previous file or buffer is still on top of the kill ring. This is important for moving text between buffers.

6.3.4. Other Kill commands

Other syntactic units can be killed, too; words, with ESC DEL (*kill-previous-word*) and ESC D (*kill-next-word*); and sentences, with ESC K (*kill-to-end-of-sentence*) and ˆX DEL (*kill-to-beginning-of-sentence*).

6.3.5. Killing by Lines

Another kill command is the ˆK command (*kill-to-end-of-line*). If issued at the beginning of a line, it kills all the text on the line, leaving it blank. If given in the middle of a line, it kills all the text up to the end of the line. If given on a line that is empty or contains only white space (blanks and tabs) the line disappears. If ˆK is done at the end of a line, it joins that line and the next line. As a consequence, if you go to the front of a non-blank line and type two ˆK's, the line disappears completely (but be careful, because one ˆK is sufficient to remove an empty line).

In general, ˆK kills from point up to the end of the line, unless it is at the end of a line, in which case it kills the line-separator 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 line-separator will be killed.

ˆK with an argument kills that many lines, including their line separators (whether the lines are empty or not). Without an argument, ˆK behaves as described in the previous paragraph. ˆU ˆK kills four lines (but note that typing ˆK four times would kill only 2 lines)

ˆK with an argument of zero kills all the text before point on the current line.

6.3.6. Appending Kills

Normally, each kill command pushes a new block onto the kill ring. However, two or more kill commands immediately in a row (without any other intervening commands) combine their text into a single entry on the ring, so that a single ˆY (*yank*) command 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.

Commands that kill forward from point add onto the end of the previously 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 needing rearrangement.

Suppose, for example you have a line containing FOO BAR BAZ with the cursor at the start of BAR. Type ESC D (*kill-next-word*), then ESC DEL (*kill-previous-word*), then ESC F (*forward-word*) to put the cursor after BAZ, and Space to insert a space. Then type ˆY (*yank*) and your line will contain BAZ FOO BAR.

6.4. The Kill Ring

To recover killed text that is no longer the most recent kill, you need the ESC Y (*yank-pop*) command. The ESC Y command can be used only immediately after a ˆY (*yank*) command or another ESC Y. It takes the yanked text inserted by the ˆY 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 ˆY to recover the last kill, and then discard it by use of ESC Y to move back to the previous one.

You can think of all the last few kills as living on a ring. After a ˆY command, the text at the front of the ring is still present in the buffer. ESC Y “rotates” the ring bringing the previous string of text to the front and this text replaces the other text in the buffer as well. Enough ESC Y commands can rotate any part of the ring to the front, so you can get at any killed text so long as it is recent enough to be still in the ring. Eventually the ring rotates all the way round and the most recently killed text comes to the front (and into the buffer) again. ESC Y with a negative argument rotates the ring backwards.

When the text you are looking for is brought into the buffer, you can stop doing ESC Y’s and the text will stay there. It is really just a copy of what’s at the front of the ring, so editing it does not change what’s in the ring. And the ring, once rotated, stays rotated, so that doing another ˆY gets another copy of what you rotated to the front with ESC Y.

If you change your mind about yanking, ˆW (*kill-region*) gets rid of the yanked text, even after any number of ESC Y’s.

7. Searching and Replacing

7.1. Searching

The search commands are useful for finding and moving to arbitrary positions in the buffer in one swift motion. For example, if you just ran the spell program on a document and you want to correct some word, you can use the search commands to move directly to that word. There are two flavors of search: *string search* and *incremental search*. The former is the default flavor — if you want to use incremental search you must rearrange the key bindings (see below).

7.1.1. Conventional Search

ˆS or ˆ\	*search-forward*	Search forward.
ˆR	*search-reverse*	Search backward.

To search for the string “FOO” you type ˆS FOO<return>. If JOVE finds FOO it moves point to the end of it; otherwise JOVE prints an error message and leaves point unchanged. ˆS searches forward from point so only occurrences of FOO after point are found. To search in the other direction use ˆR. It is exactly the same as ˆS except that it searches in the opposite direction, and if it finds the string it leaves point at the beginning of it, not at the end as in ˆS.

While JOVE is searching it displays the search string on the message line. This is so you know what JOVE is doing. When the system is heavily loaded and editing in exceptionally large buffers, searches can take several (sometimes many) seconds.

JOVE remembers the last search string you used, so if you want to search for the same string again you can type ˆS <return>. If you mistyped the last search string, you can type ˆS followed immediately by ˆR (which is *not* the *search-reverse* command in this context) which inserts the default search string, and then you can fix it up.

Note that the precise interpretation of the search string is dependent on the variable *match-regular-expressions* and is subject to the rules for *regular-expressions* to be described shortly.

7.1.2. Incremental Search

This search command is unusual in that is is *incremental*; it begins to search before you have typed the complete search string. As you type in the search string, JOVE shows you where it 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 a Return.

To use the incremental searches, you first have to bind them to suitable keys, for example to ˆS and ˆR if you want all your searching to become incremental. To do this, type

ESC X bind-to-key i-search-forward ˆS
ESC X bind-to-key i-search-reverse ˆR

The command to search is now ˆS (*i-search-forward*). ˆS reads in characters and positions the cursor at the first occurrence of the characters that you have typed so far. If you type ˆS and then F, the cursor moves in the text just after the next “F”. Type an “O”, and see the cursor move to after the next “FO”. After another “O”, the cursor is after the next “FOO”. At the same time, “FOO” has echoed on the message line.

If you type a mistaken character, you can rub it out. After the FOO, typing a DEL makes the “O” disappear from the message line, leaving only “FO”. The cursor moves back in the buffer to the “FO”. Rubbing out the “O” and “F” moves the cursor back to where you started the search.

When you are satisfied with the place you have reached, you can type a Return, 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 ˆA would exit the search and then move to the beginning of the line. Return is necessary only if the next character you want to type is a printing character, DEL, Return, or another search command, since those are the characters that have special meanings inside the search.

Sometimes you search for “FOO” and find it, but not the one you hoped to find. Perhaps there is a second FOO after the one you just found. Then type another “ˆS” and the cursor will find the next FOO. This can be done any number of times. If you overshoot, you can return to previous finds by rubbing out the “ˆS”s. Note that, in this context, “ˆS” (alternatively “ˆ\”) is a built-in use of the ˆS character and not another invocation of *i-search-forward* (which is why I have shown it in “quotes”).

After you exit a search, you can search for the same string again by typing just ˆS ˆS: one ˆS command to start the search and then another “ˆS” to mean “search again for the same string”.

If your string is not found at all, the message line says “Failing I-search”. The cursor is after the place where JOVE 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 Return or some other JOVE command to “accept what the search offered”. Or you can type ˆG, which undoes the search altogether and positions you back where you started the search.

You can also type ˆR (*i-search-reverse*) at any time to start searching backwards. If a search fails because the place you started was too late in the file, you should do this. Repeated “ˆR”s keep looking backward for more occurrences of the last search string. A “ˆS” starts going forward again. “ˆR”s can be rubbed out just like anything else.

Unlike conventional searching, incremental searching does not use the rules for regular-expressions.

7.2. Replacing

In addition to the simple Replace operation which is like that found in most editors, there is a Query Replace operation which asks, for each occurrence of the pattern, whether to replace it or not.

ESC R	*replace-string*	Replace every occurrence of the string from point to the end of the buffer.
	replace-in-region	Replace every occurrence of the string within the region.
ESC Q	*query-replace-string*	Replace occurrences of the string from point to the end of the buffer, but asking for confirmation before each replacement.

7.2.1. Global replacement

To replace every occurrence of FOO after point with BAR, you can do, ESC R FOO<return>BAR<return> (*replace-string*). Replacement takes place only between point and the end of the buffer, so if you want to cover the whole buffer you must go to the beginning first.

Another alternative is to use *replace-in-region* which is just like *replace-string* except it searches only within the region.

7.2.2. Query Replace

If you want to change only some of the occurrences of FOO, not all, then the global *replace-string* is inappropriate; instead, use, e.g., ESC Q FOO<return>BAR<return> (*query-replace-string*). This moves the cursor to each occurrence of FOO and waits for you to say whether to replace it with BAR. The things you can type when you are shown an occurrence of FOO are:

Space or Y or y
to replace the FOO with BAR.

Period
to replace this FOO and then stop.

DEL or BS or N or n
to skip to the next FOO without replacing this one.

^R or R or 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 ^X ^C (*exit-jove*) and the next FOO will be displayed.

^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 ^X ^C (*exit-jove*) and the *query-replace* will continue at the next FOO.

^U or U or u
move to the last replacement and undo all changes made on that line.

! or P or p
to replace all remaining FOO's without asking, as in *replace-string*.

Return or Q or q
to stop without doing any more replacements.

^L
redraw the screen.

7.3. Searching with Regular Expressions

When we use the searching and replacement facilities described above, JOVE can search for patterns using *regular-expressions*. The handling of regular-expressions in JOVE is somewhat like that of *ed(1)* or *vi(1)*, but with some notable additions. The precise behavior depends on the setting of the variable *match-regular-expressions*. If this variable is *on*, we use true *regular-expressions*. If it is*of f*, we have just *simple-expressions*. In what follows, the term *expression* should be interpreted as simple-expression or regular-expression according to the state of that variable.

Another variable that affects searching is *case-ignore-search*. If this variable is set to *on* then upper case and lower case letters are considered equal (except, of course, within regular-expressions such as [A−Za−z]).

Note that the rules which follow are complex, arbitrary, and different from those in other editors. Hence they may be changed significantly in future versions of JOVE.

7.3.1. Simple Regular Expressions

If the variable *match-regular-expressions* is *off*, the search pattern is interpreted as follows:

^ (at the start of a pattern or sub-pattern)
Matches the empty string at the beginning of a line.

$ (at the end of a pattern or sub-pattern)
Matches the empty string at the end of a line.

\< Matches the empty string at the beginning of a word. What makes up a word depends on the major mode of the buffer that you are searching in. In all modes a word is a contiguous sequence of characters which have some defined pattern, bounded by characters that don't fit that pattern or by the beginning or end of the line. The individual modes' word patterns are as follows:

Fundamental	upper and lower case letters and digits.
Text	upper and lower case letters and digits plus apostrophe (').
C	upper and lower case letters and digits plus "$" and "_" (underscore).
Lisp	upper and lower case letters and digits plus "!$%&*+−/:<=>?^_{|}~" and Delete.

\> Matches the empty string at the end of a word.

c Matches the character *c* where *c* is not one of <, >, (,), {, } or |. In particular, \^, \$ and \\ match the characters ^, $ and \. When full regular-expressions are in use, \., * and \[will also be required.

c — Matches the character *c* where *c* is not \ or ^ (at the start of a pattern) or $ (at the end of a pattern) (plus a few further things if *match-regular-expressions* is *on*).

\{*c1...cN*\} — Matches whatever the sequence of regular-expressions *c1..cN* would have matched. Note that full regular-expression capability (even the \| construct described below) is provided within \{...\} whatever the setting of the variable *match-regular-expressions*. \{...\} provides a grouping construct like parentheses in algebraic expressions. Thus “aa\{xx\|yy\}bb” searches for “aaxxbb” or “aayybb”.

\(*c1..cN*\) — Matches whatever the sequence of expressions *c1..cN* would have matched, where the expressions are any of those described above (and also the additional full regular-expressions if *match-regular-expressions* is *on*). This is used to tag part of the search text for later reference via **n** (see below). \(*c1..cN*\) patterns may be nested. Observe that use of the \| construct (see below) directly within a \(...\) is precluded.

n — Matches the *n*’th \(*c1..cN*\) pattern where *n* is between 1 and 9. The \(*c1..cN*\) patterns are numbered by counting the \(sequences starting from the beginning of the search pattern, resetting to 1 (or to the value at the start of an enclosing \{...\}) whenever a \| is encountered. To avoid confusion in the counting, it is required that each alternative (separated by \|) within a \{...\} should contain the same number of \(...\)s. For example, the search pattern “^\(\{ab\|cd\}\)\1$” searches for all non-empty lines which contain just “abab” or “cdcd” (but not “abcd”). It is an error in the search pattern to reference a \(*c1..cN*\) pattern that follows the occurrence of *n*.

c1..cN — Matches the longest string matched by *c1*, followed by the longest string matched by *c2*, and so on. The expressions *c1..cN* are any of those described above (and also the additional full regular-expressions if *match-regular-expressions* is *on*).

c1..cN\|*d1..dN* — Matches the longest string matched by *c1..cN*, if any, and otherwise the longest string matched by *d1..dN*. Multiple \| sequences may be used to indicate more alternatives. The sequences*c1..cN* and *d1..dN* are any of those described above, which means that \| has lower precedence than any of the other operators. Each alternative must have the same number of \(...\) groups, as already explained. Thus, “\<foo\|bar\|baz\>” matches any word beginning with “foo”, any occurrence of the string “bar”, or any word ending in “baz”.

In the replacement string:

n — Is replaced with the characters matched by the *n*’th \(*c1..cN*\) in the search pattern where *n* is between 1 and 9. For example, one could replace “\<\(\{FOO\|BAR\|BAZ\}\)\>” with “[\1]” to enclose every occurrence of the words FOO, BAR and BAZ within [...].

\0 — Is replaced with the characters matched by the entire search pattern.

c — Inserts the character *c* where *c* is not a digit.

c — Inserts the character *c* where *c* is not \.

7.3.2. Full Regular Expressions

If the variable *match-regular-expressions* is *on*, the following additional special matching rules are used. Observe that special meanings now attach to the characters **.**, ∗ and [, which can therefore no longer stand for themselves.

In the search pattern:

c — Matches the character *c* where *c* is not one of **.**, ∗, [, \, ^ (at the start of a line) or $ (at the end of a line).

. — Matches any character, but not a line-separator.

[*c1..cN*] — Matches any of the characters in the sequence of characters *c1..cN* provided circumflex (^) is not the first character of the sequence (see below). The only special characters recognized while parsing the sequence are “]”, “−” and “\”. All may be represented by escaping them with a backslash (\): “\]”, “\−”, “\\”. Ranges of characters may be indicated by *a*−*b* where *a* is the first character of the range and *b* is the last. The special meaning of − is lost if it appears as the first or last character of the sequence. The special meaning of] is lost if it appears as the first character of the sequence.

[^*c1..cN*]	Matches any character except those contained in the sequence of characters *c1..cN*. The circumflex (^) is not special except immediately following the left bracket.
*c**	Matches zero or more occurrences of the expression *c*. The expression *c* may be any of the expressions covered above except for ^ and $ (which match null strings), \(*c1..cN*\) and *c1..cN*\\|*d1..dN* (which would not work), and \{*c1...cN*\} (arbitrarily forbidden).

8. Commands for English Text

JOVE has many commands that work on the basic units of English text: words, sentences and paragraphs.

8.1. Word Commands

JOVE has commands for moving over or operating on words. By convention, they are all ESC commands.

ESC F	*forward-word*	Move Forward over a word.
ESC B	*backward-word*	Move Backward over a word.
ESC D	*kill-next-word*	Kill forward to the end of a word.
ESC DEL	*kill-previous-word*	Kill backward to the beginning of a word.

Notice how these commands form a group that parallels the character-based commands, ^F, ^B, ^D, and DEL.

The commands ESC F and ESC B move forward and backward over words. They are thus analogous to ^F and ^B, which move over single characters. Like their Control- analogues, ESC F and ESC B move over several words if given an argument (and in the opposite direction with negative arguments). Forward motion stops right after the last letter of the word; backward motion stops right before the first letter.

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

ESC DEL kills the word before point. It kills everything from point back to where ESC B would move to. If point is after the space in “FOO, BAR”, then “FOO, ” is killed. If you wish to kill just “FOO”, then do an ESC B and an ESC D instead of an ESC DEL.

Note that the term “word” in all of these commands refers simply to a sequence of upper and lower case letters and digits. It is not dependent on the major mode of the buffer as was the case with regular-expressions involving \< and \>. Thus it will require two uses of ESC D to remove a word such as “isn’t”, even if the major mode is Text mode.

8.2. Sentence Commands

The JOVE commands for manipulating sentences and paragraphs are mostly ESC commands, so as to resemble the word-handling commands.

ESC A	*backward-sentence*	Move back to the beginning of the sentence.
ESC E	*forward-sentence*	Move forward to the end of the sentence.
ESC K	*kill-to-end-of-sentence*	Kill forward to the end of the sentence.
^X DEL	*kill-to-beginning-of-sentence*	Kill back to the beginning of the sentence.

The commands ESC A and ESC E move to the beginning and end of the current sentence, respectively. They were chosen to resemble ^A and ^E, which move to the beginning and end of a line. Unlike them, ESC A and ESC E if repeated or given numeric arguments move over successive sentences. JOVE considers a sentence to end wherever there is a “.”, “?”, or “!” followed by the end of a line or by one or more spaces. Neither ESC A nor ESC E moves past the end of the line or the spaces which delimit the sentence.

Just as ^A and ^E have a kill command, ^K, to go with them, so ESC A and ESC E have a corresponding kill command ESC K 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. Positive arguments serve as a repeat count.

There is a special command ^X DEL for killing back to the beginning of a sentence, because this is useful when you change your mind in the middle of composing text.

8.3. Paragraph Commands

The JOVE commands for handling paragraphs are

	backward-paragraph	Move back to the start of the previous paragraph.
ESC]	*forward-paragraph*	Move forward to the end of the next paragraph.

Note that "ESC [" is not bound to *backward-paragraph*, as might have been expected, on most (i.e. ANSI-compliant) terminals because that sequence is used as a prefix for codes generated by the Function Keys.

Backward-paragraph moves to the beginning of the current or previous paragraph, while *forward-paragraph* moves to the end of the current or next paragraph. Paragraphs are delimited by lines of differing indent, or lines with text formatter commands, or blank lines. JOVE knows how to deal with most indented paragraphs correctly, although it can get confused by one- or two-line paragraphs delimited only by indentation.

8.4. Text Indentation Commands

Tab	*handle-tab*	Indent "appropriately" in a mode-dependent fashion.
Linefeed	*newline-and-indent*	Is the same as Return, except it copies the indent of the line you just left.
ESC M	*first-non-blank*	Moves to the line's first non-blank character.

The way to request indentation is with the Tab command. Its precise effect depends on the major mode. In *Text* mode, it indents to the next tab stop (as determined by the variable *tab-width*, whose default value is 8). In *C* mode or *Lisp* mode, it indents to the "right" position for those programs (see later).

To move over the indentation on a line, do ESC M (*first-non-blank*). This command, given anywhere on a line, positions the cursor at the first non-blank, non-tab character on the line.

8.5. Text Filling

	auto-fill-mode	Toggle the minor mode *auto fill*.
ESC J	*fill-paragraph*	Refill the paragraph containing the cursor.
	fill-region	Refill the region.
	fill-comment	Refill a comment, depending on the major mode.
	left-margin-here	Sets the variable *left-margin* from point.
	right-margin-here	Sets the variable *right-margin* from point.

Auto Fill mode is a minor mode that causes text to be *filled* (broken up into lines that fit in a specified width) automatically as you type it in. If you alter existing text so that it is no longer properly filled, JOVE can fill it again if you ask.

Entering *Auto Fill* mode is done with ESC X *auto-fill-mode*. From then on, lines are broken automatically at spaces when they get longer than the desired width. To leave *Auto Fill* mode, once again execute ESC X *auto-fill-mode*. When *Auto Fill* mode is in effect, the word **Fill** appears in the mode line.

If you edit the middle of a paragraph, it may no longer be filled correctly. To refill a paragraph, use the command ESC J (*fill-paragraph*). It causes the paragraph that point is inside to be filled. All the line breaks are removed and new ones inserted where necessary. Similarly, *fill-region* may be used to refill a region other than a paragraph. The special command *fill-comment* is only meaningful in those major modes, currently C mode and Lisp mode, which support it.

The maximum line width for filling is in the variable *right-margin*. Both ESC J and auto-fill make sure that no line exceeds this width. The value of *right-margin* is initially 78.

Normally ESC J figures out the indent of the paragraph and uses that same indent when filling. If you want to force some other indent for a paragraph, you set *left-margin* to the new position and type ˆU ESC J, since *fill-paragraph* uses the value of *left-margin* when supplied with a numeric argument.

If you know where you want to set the variable *right-margin* but you don't know the actual value, move to where you want to set the value and use the *right-margin-here* command. *left-margin-here* does the same for the *left-margin* variable.

8.6. Case Conversion Commands

ESC L	*case-word-lower*	Convert the following word to lower case.
ESC U	*case-word-upper*	Convert the following word to upper case.
ESC C	*case-word-capitalize*	Capitalize the following word.
	case-character-capitalize	Capitalize the character after point.
	case-region-lower	Convert the region to lower case.
	case-region-upper	Convert the region to upper case.

The word conversion commands are most useful. ESC L converts the word after point to lower case, moving past it. Thus, successive ESC L's convert successive words. ESC U converts to all capitals instead, while ESC C 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 ESC L, ESC U or ESC C on each word as appropriate.

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 the cursor, treating it as a whole word.

8.7. Commands for Fixing Typos

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

DEL	*delete-previous-character*	Delete last character.
ESC DEL	*kill-previous-word*	Kill last word.
ˆX DEL	*kill-to-beginning-of-sentence*	Kill to beginning of sentence.
ˆT	*transpose-characters*	Transpose two characters.
ˆX ˆT	*transpose-lines*	Transpose two lines.
ESC Minus ESC L		Convert last word to lower case.
ESC Minus ESC U		Convert last word to upper case.
ESC Minus ESC C		Convert last word to lower case with initial capital.

8.7.1. Killing Your Mistakes

The DEL command is the most important correction command. When used among printing (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 ESC DEL or ˆX DEL. ˆX DEL is particularly useful when you are thinking of what to write as you type it, in case you change your mind about phrasing. ESC DEL and ˆX DEL save the killed text for subsequent yanking.

ESC 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 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.

8.7.2. Transposition

The common error of transposing two characters can be fixed with the ˆT (*transpose-characters*) command. Normally, ˆT transposes the two characters on either side of the cursor and moves the cursor forward one character. Repeating the command several times "drags" a character to the right. When given at the end of a line, rather than switching the last character of the line with the line-separator, which would be useless, ˆT transposes the last two characters on the line. So, if you catch your transposition error right away, you can fix it with just a ˆT. If you don't catch it so quickly, you must move the cursor back to between the two characters.

To transpose two lines, use the ˆX ˆT (*transpose-lines*) command. The line containing the cursor is exchanged with the line above it; the cursor is left at the beginning of the line following its original position.

8.8. Checking and Correcting Spelling

When you write a paper, you should correct its spelling at some point close to finishing it. To correct the entire buffer, do ESC X *spell-buffer*. This invokes the *spell* program, which prints a list of all the misspelled words. JOVE catches the list and places it in a JOVE buffer called **Spell**. You now edit the **Spell** buffer (technically, you are in a recursive edit at this point) by deleting from that buffer any words that aren't really errors. Next, type ^X ^C (*exit-jove*) to escape from the recursive edit, and JOVE now positions you at the first misspelled word in the original buffer. Correct that mistake with the usual editor commands. Then you can go forward to each other misspelled word with ^X ^N (*next-error*) or backward with ^X ^P (*previous-error*). If, in the course of editing a mistake, you get completely lost, the command *current-error* will put you back at the error you were supposed to be correcting.

9. Buffers

When we speak of "the buffer", which contains the text you are editing, we may have given the impression that there is only one. In fact, there may be many of them, each with its own body of text. At any time only one buffer can be *current* and available for editing, but it is easy to switch to a different one. Each buffer individually remembers which file it contains, what modes are in effect, and whether there are any changes that need saving.

^X B	*select-buffer*	Select or create a buffer.
^X ^B	*list-buffers*	List the existing buffers.
^X K	*delete-buffer*	Delete the contents of a buffer and destroy it.
	erase-buffer	Delete the contents of a buffer.
	kill-some-buffers	Destroy unwanted buffers.
	rename-buffer	Rename the selected buffer.
	buffer-position	Report the position of point within the buffer.
ESC ~	*make-buffer-unmodified*	Tell JOVE to forget that the buffer has been changed.
^X ^F	*find-file*	Read a file into its own buffer.
^X ^S or ^X \	*save-file*	Save the selected buffer.
^X ^M	*write-modified-files*	Save all modified buffers.

Each buffer in JOVE has a single name which normally doesn't change. A buffer's name can be any length. The name of the currently selected buffer and the name of the file contained in it are visible in the mode line of any window displaying that buffer. A newly started JOVE has only one buffer, named **Main**, unless you specified files to edit in the shell command that started JOVE.

9.1. Creating and Selecting Buffers

To create a new buffer, you need only think of a name for it (say, FOO) and then type ^X B FOO<return> (*select-buffer*). This makes a new, empty buffer (if one by that name didn't exist previously) and selects it for editing. The new buffer does not contain any file, so if you try to save it you will be asked for the filename to use. Each buffer has its own major mode; the new buffer's major mode is Text mode by default.

To return to buffer FOO later after having switched to another, the same command ^X B FOO<return> is used, since ^X B can tell whether a buffer named FOO exists already or not. Just ^X B<return> reselects the previous buffer. Repeated ^X B<return>'s alternate between the last two buffers selected.

9.2. Using Existing Buffers

To get a list of all the buffers that exist, type ^X ^B (*list-buffers*). Each buffer's type, name, and contained filename is printed. An asterisk before the buffer name indicates that there are changes that have not yet been saved. The number that appears at the beginning of a line in a ^X ^B listing is that buffer's *buffer number*. You can select a buffer by typing its number in place of its name. If a buffer with that number doesn't already exist, a new buffer is created with that number as its name.

If several buffers have modified text in them, you can save them with ^X ^M (*write-modified-files*). This finds all the buffers that need saving and then saves them. Saving the buffers this way is much easier and more efficient (but more dangerous) than selecting each one and typing ^X ^S (*save-file*). If you give ^X ^M an argument, JOVE will ask for confirmation before saving each buffer.

ESC X *rename-buffer* <new name><return> changes the name of the selected buffer.

ESC X *erase-buffer* <buffer name><return> erases the contents of <buffer name> without destroying it entirely.

ESC X *buffer-position* reports the position of point within the selected buffer, both as lines/total-lines, chars/total-chars, and as a percentage.

Sometimes you will change a buffer by accident. Even if you undo the effect of the change by editing, JOVE still knows that "the buffer has been changed". You can tell JOVE to pretend that there have been no changes with the ESC ~ command (*make-buffer-unmodified*). This command simply clears the "modified" flag which says that the buffer contains changes which need to be saved. Even if the buffer really *is* changed JOVE will still act as if it were not.

9.2.1. Killing Buffers

After you use a JOVE for a while, it may fill up with buffers which you no longer need. Eventually you can reach a point where trying to create any more results in an "out of memory" or "out of lines" error. When this happens you will want to destroy some buffers with the ˆX K (*delete-buffer*) command. You can destroy the buffer FOO by doing ˆX K FOO<return>. If you type ˆX K <return> JOVE will kill the previously selected buffer. If you try to kill a buffer that needs saving JOVE will ask you to confirm it.

If you need to kill several buffers, use the command *kill-some-buffers*. This prompts you with the name of each buffer and asks for confirmation before killing it.

10. File Handling

The basic unit of stored data is the file. Each program, each document, lives usually in its own file. To edit a program or document, the file that contains it must first be brought into a buffer, either an existing one (*visit-file*) or one created specifically for that file (*find-file*). To make your changes to the file permanent on disk, you must *save* the file.

10.1. Reading Files

ˆX ˆF	*find-file*	Read a file into its own buffer.
ˆX ˆV	*visit-file*	Visit a file.
ˆX ˆR	*visit-file*	An alternative to ˆX ˆV.
ˆX ˆI	*insert-file*	Insert a file at point.

JOVE remembers the name of the file that is contained in each buffer (remember the ˆX ˆB (*list-buffers*) command). The name of the buffer is visible in its mode line together with the name of its file.

You can read a file into its own newly created buffer by typing ˆX ˆF (*find-file*), followed by the filename. The name of the new buffer will be the last element of the file's pathname. You can abort the command by typing ˆG, or edit the filename with any of the standard JOVE commands (e.g., ˆA, ˆE, ˆF, ESC F, ESC DEL). If the filename you wish to visit is similar to the filename in the current mode line (the default filename), you can type ˆR to insert the default and then edit it. For more about this and other special methods of constructing filenames, see the sections on *The Message Line* and *Name Completion* earlier in this manual. When you are satisfied type Return, and the new file's text will appear on the screen, and its name in the mode line.

The ˆF in ˆX ˆF stands for "Find", because if the specified file already resides in some buffer, that buffer is simply re-selected. So you need not remember whether you have brought the file in already or not. A buffer created by ˆX ˆF can be reselected later with ˆX B or ˆX ˆF, whichever you find more convenient.

Visiting a file means copying its contents into an existing buffer so that you can edit them. To visit a file, use the command ˆX ˆV or ˆX ˆR (*visit-file*), followed by the filename. The name of the new file will appear in the mode line but the name of the buffer will be unchanged.

If you alter one file and then visit another in the same buffer, JOVE offers to save the old one. If you answer YES (or y), the old file is saved; if you answer NO (or n), all the changes you have made to it since the last save are lost. You should not type ahead after a file visiting command, because your type-ahead might answer an unexpected question in a way that you would regret.

ˆX ˆI (*insert-file*) followed by a filename reads the file and inserts it into the buffer at point, leaving point before the file contents and the mark at their end.

The changes you make with JOVE are made in a copy inside JOVE. The file itself is not changed. The changed text is not permanent until you *save* it in a file. The first time you change the text, an asterisk appears in the mode line; this indicates that the text contains fresh changes which will be lost unless you save them.

What if you want to create a file? Just read it with *find-file* or *visit-file*. JOVE prints *(New file)* but aside from that behaves as if you had read an existing empty file. If you make any changes and save them, the file is created then. If you read a nonexistent file unintentionally (because you typed the wrong filename), go ahead and read the file you meant. The unwanted file will not have been created.

10.2. Writing files

ˆX ˆS or ˆX ˆ\	*save-file*	Save the file in the selected buffer.
ˆX ˆW	*write-file*	Write the selected buffer to a different file.
	write-region	Write the region to the specified file.
	append-region	Append the region to the specified file.

If you wish to save the file and make your changes permanent, type ˆX ˆS. After the save is finished, ˆX ˆS prints the filename and the number of characters and lines that it wrote to the file. If there are no changes to save (no asterisk in the mode line), the file is not saved; otherwise the changes are saved and the asterisk in the mode line disappears.

If JOVE is about to save a file and sees that the date of the version on disk does not match what JOVE last read or wrote, JOVE notifies you of this fact, and asks what to do, because this probably means that something is wrong. For example, somebody else may have been editing the same file. If this is so, there is a good chance that your work or his work will be lost if you don't take the proper steps. You should first find out exactly what is going on. If you determine that somebody else has modified the file, save your file under a different filename and then DIFF the two files to merge the two sets of changes. (The “patch” command is useful for applying the results of context diffs directly). Also get in touch with the other person so that the files don't diverge any further.

ˆX ˆW <filename><return> (*write-file*) writes the contents of the buffer into the file <filename>, changing the name of the file recorded in the mode line accordingly. It can be thought of as a way of “changing the name” of the file in the buffer. Unlike ˆX ˆS, *write-file* saves even if the buffer has not been changed.

ESC X *write-region* <file><return> writes the region (the text between point and mark) to the specified file. It does not change the buffer's filename.

ESC X *append-region* <file><return> appends the region to <file>. The text is added to the end of <file>.

10.3. How to Undo Drastic Changes to a File

If you have made several extensive changes to a file and then change your mind about them, and you haven't yet saved them, you can get rid of them by reading in the previous version of the file. You can do this with the ˆX ˆV (*visit-file*) command, to visit the unsaved version of the file. Remember to tell it not to save your existing changes when it asks.

11. Windows

11.1. Multiple Windows

JOVE allows you to split the screen into two or more *windows* and use them to display parts of different buffers, or different parts of the same buffer.

```
#define getchar()    getc(stdin)
#define putchar(x)   putc((x), stdout)               < first Window
JOVE (C RO)  [stdio.h:1]  "/usr/include/stdio.h"  --   < the Mode Line
{
    printf("Hello world!\n");
    return 0;                                          < second Window
}□
JOVE (C OvrWt)  [Main:1]  "aloha.c"  --  /home/foo     < the Mode Line
[Point pushed]                                         < the Message Line
```

ˆX 2	*split-current-window*	Divide the active window into two smaller ones.
ˆX 1	*delete-other-windows*	Delete all windows but the current one.
ˆX D	*delete-current-window*	Delete the active window.
ˆX N	*next-window*	Switch to the next window.
ˆX P	*previous-window*	Switch to the previous window.
ˆX O	*previous-window*	Same as ˆX P.
ˆX ˆ	*grow-window*	Make this window bigger.
	shrink-window	Make this window smaller.
ESC ˆV	*page-next-window*	Scroll the other window.
ˆX 4	*window-find*	Combination window command.

When using *multiple window* mode, the window portion of the screen is divided into *windows*, which can display different pieces of text. Each window can display different buffers, or different parts of the same buffer. Only one of the windows is *active*, viz. the window which the cursor is in. Editing normally takes place in that window alone. To edit in another window, you would give a command to move the cursor to the other window, and then edit there.

Each window includes a mode line for the buffer it is displaying. This is useful to keep track of which window corresponds with which buffer and which file. In addition, the mode line serves as a separator between windows. Normally, the variable *mode-line-should-standout* is *on* so that JOVE displays the mode-line in reverse video (assuming your particular terminal has the reverse video capability). However, if the variable *scroll-bar* is also *on*, a portion of the mode line is left clear to indicate how the window is located within the buffer.

The command ˆX 2 (*split-current-window*) divides the active window into two. A new mode line appears across the middle of the original window, dividing its display area into two halves. Both windows contain the same buffer and display the same position in it, namely where point was at the time you issued the command. The cursor moves to the second window.

To return to viewing only one window, use the command ˆX 1 (*delete-other-windows*). The active window expands to fill the whole screen, and the other windows disappear until the next ˆX 2. (The buffers and their contents are unaffected by any of the window operations).

While there is more than one window, you can use ˆX N (*next-window*) to switch to the next window, and ˆX P (*previous-window*) to switch to the previous one. If you are in the bottom window and you type ˆX N, you will be placed in the top window, and the opposite thing happens when you type ˆX P in the top window. ˆX O is the same as ˆX P. It stands for “other window” because, when there are only two windows, repeated use of this command will switch between them.

Often you will be editing one window while using the other just for reference. Then, the command ESC ˆV (*page-next-window*) is very useful. It scrolls the next window up, just as if you switched to the next window, typed ˆV, and switched back. With a negative argument, ESC ˆV will scroll down.

When a window splits, both halves are approximately the same size. You can redistribute the screen space between the windows with the ˆX ˆ (*grow-window*) command. It makes the active window grow one line bigger, or as many lines as is specified with a numeric argument. Use ESC X*shrink-window* to mak e the active window smaller.

11.2. Multiple Buffers in Multiple Windows

Buffers can be selected independently in each window. The ˆX B (*select-buffer*) command selects a different buffer in the active window (i.e. the one containing the cursor). Other windows' buffers do not change. Likewise, the ˆX ˆF (*find-file*) command reads a new file into a new buffer in the active window.

You can view the same buffer in more than one window. Although the same buffer appears in both windows, they have different values of point, so you can move around in one window while the other window continues to show the same text. If you make changes in one window, and the same place in the buffer happens to be visible in the other window, your changes will appear simultaneously in both places.

If you have the same buffer in both windows, you must beware of trying to visit a different file in one of the windows with ˆX ˆV, because if you bring a new file into this buffer it will replace the old file in *both* windows. To view different files in different windows, you must switch buffers in one of the windows first (with ˆX B) or use ˆX ˆF (*find-file*).

A convenient "combination" command for viewing something in another window is ˆX 4 (*window-find*). With this command you can ask to see any specified buffer, file or tag in the other window. Follow the ˆX 4 with either B and a buffer name, F and a filename, or T and a tag name. This switches to the other window and finds there what you specified. If you were previously in one-window mode, multiple-window mode is entered. ˆX 4 B is similar to ˆX 2 ˆX B; ˆX 4 F is similar to ˆX 2 ˆX ˆF; ˆX 4 T is similar to ˆX 2 ˆX T. The difference is one of efficiency, and also that ˆX 4 works equally well if you are already using two windows.

11.3. Controlling the Display

Since only part of a large file will fit in a window, JOVE tries to show the portion that is likely to be interesting. The display control commands allow you to bring a different portion of the buffer within the active window.

ˆL	*redraw-display*	Reposition point at a specified vertical position, OR clear and redraw the window with point in the same place.
ESC ˆL	*clear-and-redraw*	Clear and redraw the entire screen.
ˆV	*next-page*	Scroll forwards (a page or a few lines).
ESC V	*previous-page*	Scroll backwards.
ˆZ	*scroll-up*	Scroll forward some lines.
ESC Z	*scroll-down*	Scroll backwards some lines.
	scroll-left	Scroll the window to the left.
	scroll-right	Scroll the window to the right.
	number-lines-in-window	Number the lines in the window.

A window is rarely large enough to display all of your file. If the whole buffer doesn't fit on the screen, JOVE shows a contiguous portion of it, containing point. It continues to show approximately the same portion until point moves outside of what is displayed; then JOVE chooses a new portion centered around a new point. This is JOVE's guess as to what you are most interested in seeing, but if the guess is wrong, you can use the display control commands to see a different portion.

If the window holds only a part of the buffer, and if the variable *scroll-bar* is *on*, a clear patch in the (otherwise reverse-videoed) mode line indicates what proportion is visible. This is especially useful for mouse-based versions of the editor, such as *xjove*.

First we describe how JOVE chooses a new window position on its own. The goal is usually to place point half way down the window. This is controlled by the variable *scroll-step*, whose value is the number of lines above the bottom or below the top of the window that the line containing point is placed. A value of 0 (the initial value) means center point in the window.

The basic display control command is ˆL (*redraw-display*). In its simplest form, with no argument, it tells JOVE to choose a new portion of the buffer, centering the existing point half way from the top as usual. ˆL with a positive argument chooses a new portion so as to put point that many lines 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.

If during the ˆL command point stays on the same line, the window is first cleared and then redrawn. Thus, two ˆL's in a row are guaranteed to clear and redraw the active window. ESC ˆL (*clear-and-redraw*) will clear and redraw the entire screen.

The *scrolling* commands ˆV, ESC V, ˆZ, and ESC Z let you move the whole display up or down a few lines. In fact, with a numeric argument, ˆV is identical to ˆZ and ESC V to ESC Z. So ˆV (*next-page*) or ˆZ (*scroll-up*) with an argument shows you that many more lines at the bottom of the screen, moving the text and point up together as ˆL might. ˆV or ˆZ with a negative argument shows you more lines at the top of the screen, as does ESC V (*previous-page*) or ESC Z (*scroll-down*) with a positive argument.

ˆV with no argument scrolls the buffer a window at a time. It takes the last line at the bottom of the window and puts it at the top, followed by nearly a whole window of lines not visible before. Point is put at the top of the window. Thus, each ˆV shows the "next page of text", except for one line of overlap to provide context. To move backward, use ESC V without an argument, which moves a whole window backwards (again with a line of overlap).

With no argument, ˆZ and ESC Z scroll one line forward and one line backward, respectively. These are convenient for moving in units of one line without having to type a numeric argument.

The commands *scroll-left* and *scroll-right* scroll the entire window in the specified direction by the amount of the argument (or for 10 characters by default). The argument may be negative.

The command *number-lines-in-window* causes each line displayed to be preceded by its line-number (and giving the command again restores the former state). Note that this state is a property of the window, not of the buffer.

12. Processes Under JOVE

An important feature of JOVE is its ability to interact with You can run commands from JOVE and catch their output in JOVE buffers. Two mechanisms are provided, *interactive processes* and *non-interactive processes*.

12.1. Interactive Processes

With most modern systems, JOVE has the capability of running interactive processes, accepting your input and capturing your output in a buffer.

shell	Run a shell in an interactive process buffer.
i-shell-command	Run a UNIX command in an interactive process buffer.

12.1.1. How to Run a Shell in a Window

Type ESC X *shell*<return> to start up a shell. JOVE will create a buffer, called ∗**shell**∗**−1**, and choose a window for this new buffer. The shell process is now said to be attached to the buffer. The program that is now running in the buffer is that specified by the variable *shell*, which is itself initialized from your SHELL environment variable. The shell command is called with the flag "-c", or whatever else the variable *shell-flags* has been set to.

Use an argument (**n**) with the *shell* command to create other buffers (∗**shell**∗**−n**) running independent shells.

Once an interactive process is running you can select another buffer into that window, or you can delete that window altogether. You can go off and do some other editing while the command is running. This is useful for commands that do sporadic output and run for fairly long periods of time. When you reselect that buffer later it will be up to date. That is, even though the buffer wasn't visible it was still receiving output from the process. You don't have to worry about missing anything when the buffer isn't visible.

12.1.2. How to Run a Command in a Window

To run a command interactively from JOVE type ESC X *i-shell-command* <command-name><return>. For example, to run the desk calculator, you do:

```
ESC X i-shell-command dc<return>
```

Then JOVE picks a buffer in which the output from the command will be placed, named after the command (*dc* in this case). Compare this command to the non-interactive *shell-command* to be described presently.

12.1.3. Facilities available in interactive windows

What you type into an interactive process isn't seen immediately by the process; instead JOVE waits until you type an entire line before passing it on to the process to read. This means that before you type Return all of JOVE's editing capabilities are available for fixing errors on your input line. If you discover an error at the beginning of the line, rather than erasing the whole line and starting over you can simply move to the error, correct it, move back, and continue typing.

In fact Return does different things depending on both your position in the buffer and on the state of the process. In the normal case, when point is in the last line of the buffer, Return does as already described: it inserts a line-separator and then sends the line to the process. If you are somewhere else in the buffer, possibly positioned at a previous command that you want to edit, Return will place a copy of that line at the end of the buffer and move you there (the prompt will be discarded if there is one — the variable *process-prompt* specifies what to discard) Then you can edit the line and type Return as in the normal case. If the process has died for some reason, Return does nothing. It doesn't even insert itself. If that happens unexpectedly, you should type ESC X *list-processes*<return> to get a list of each process and its state. If your process died abnormally, *list-processes* may help you figure out why.

Another feature is that you have the entire history of your session in the JOVE buffer. You don't have to worry about output from a command moving past the top of the screen. If you missed some output you can move back through it with ESC V and other commands. In addition, you can save yourself retyping a command (or a similar one) by sending edited versions of previous commands, or edit the output of one command to become a list of commands to be executed ("immediate shell scripts").

There are several special commands available only in interactive windows.

Return	*process-newline*	Send a line to a process.
	process-send-data-no-return	Send a line to a process, but without the line-separator.
ˆC ˆC	*interrupt-process*	Send SIGINT to the process.
ˆC \	*quit-process*	Send SIGQUIT to the process.
ˆC ˆZ	*stop-process*	Send SIGTSTP to the process.
ˆC ˆY	*dstop-process*	Send SIGTSTP when next the process tries to read input.
	continue-process	Send SIGCONT to the process.
	kill-process	Send SIGKILL to the process in a specified buffer.
ˆC ˆD	*eof-process*	Send EOF to the process.

Although Return is automatically bound to *process-newline*, the various ˆC ... must be explicitly bound in your (or your system administrator's) customization. The effects of Return (*process-newline*) in various circumstances have already been described above. The effects of ˆC **ˆx** for various **x** are equivalent to sending **ˆx** to the shell, assuming the customary bindings as set up by *stty*. Observe that **ˆx** without a preceding ˆC will have some other effect in JOVE (for example, ˆD is still bound to *delete-next-character*).

12.1.4. DBX in interactive windows

If the debugging program *dbx* is provided with your system, you may of course run it in an interactive window. Before doing this, you should turn on the dbx minor mode. The effect of this is that, every time *dbx* halts with a message line specifying a filename and linenumber (at every breakpoint, for example), *find-file* will automatically be called on that filename, it will appear in another window, and point will be moved to that line. Thus you may easily follow the progress of the program being debugged.

12.2. Non-interactive Processes

The reason these are called non-interactive processes is that you can't type any input to them; you can't interact with them; they can't ask you questions because there is no way for you to answer. Remember that JOVE (not the process in the window) is listening to your keyboard, and JOVE waits until the process dies before it looks at what you type.

ˆX !	*shell-command*	Run a UNIX command in a buffer.
	shell-command-no-buffer	Run a UNIX command without any buffer.
	shell-command-to-buffer	Run a UNIX command in a named buffer.
	shell-command-with-typeout	Run a UNIX command sending output to the screen.

To run a command from JOVE just type ˆX ! <command-name><return>. For example, to get a list of all the users on the system, you do:

```
ˆX ! who<return>
```

Then JOVE picks a buffer in which the output from the command will be placed, named after the command. E.g., "who" uses a buffer called **who**; "ps alx" uses **ps**; and "egrep -n foo ∗.c" uses**egr ep**. If JOVE wants to use a buffer that already exists it first erases the old contents. If the buffer it selects holds a file, not output from a previous shell command, you must first delete that buffer with ˆX K (*delete-buffer*). There are variants of the command where there is no buffer, where you can name your own buffer, and where the output is direct to the screen (see the section on typeout at the start of this manual).

Once JOVE has picked a buffer it puts that buffer in a window so you can see the command's output as it is running. If there is only one window JOVE will automatically make another one. Otherwise, JOVE tries to pick the most convenient window other than the current one.

It is not a good idea to type anything while the command is running because JOVE won't see the characters (and thus won't execute them) until the command finishes, so you may forget what you have typed. If you really want to carry on with other editing tasks while it is running, it is better to use the *i-shell-command* described previously.

If you want to interrupt the command for some reason (perhaps you mistyped it, or you changed your mind) you can type ˆ] (or whatever else has been put in the variable *interrupt-character*). Typing this inside JOVE while a process is running is the same as typing ˆC when you are outside JOVE, namely the process is interrupted.

When the command finishes, JOVE puts you back in the window in which you started. Then it prints a message indicating whether or not the command completed successfully in its (the command's) opinion. That is, if the command had what it considers an error (or you interrupt it with ˆ]) JOVE will print an appropriate message.

12.2.1. Applications of Non-Interactive Processes

ˆX ! (*shell-command*) is useful for running commands that do some output and then exit. So you could type ˆX ! spell <filename> and it would create a buffer "spell", fill it with all the spelling mistakes in <filename> and display it in a window. However, as we have already seen, there is a built-in JOVE command to do this job (and more) which, behind the scenes, issues exactly that *shell-command*. Thus, the built in usage of this facility by JOVE itself is as important as any use you might invent for yourself.

You could run a program through a compiler using *shell-command*, but again JOVE provides a special command for the job. This is the ˆX ˆE (*compile-it*) command. If you run *compile-it* with no argument it runs the *make* program into a buffer. If you need a special command or want to pass arguments to *make*, run *compile-it* with any argument (ˆU is good enough) and you will be prompted for the command to execute. If any error messages are produced, they are treated specially by JOVE. That treatment is the subject of the next section.

Another useful example of using the *shell-command* would be to type ˆX ! egrep -l <identifier>*.c, to gi ve you a list of all your .c files containing that <identifier>.

12.2.1.1. Error Message Parsing

	parse-errors	Prepare to exhibit the errors listed in the buffer.
	parse-spelling-errors-in-buffer	Prepare to exhibit the listed spelling errors.
ˆX ˆN	*next-error*	Move to the next listed error.
ˆX ˆP	*previous-error*	Move to the previous listed error.
	current-error	Move to the current listed error.

When you have your error messages in a buffer as produced by the *shell-command*, you run the *parse-errors* command (this happens automatically after a *compile-it*). Each line in this buffer should specify a filename and a linenumber (JOVE knows how to interpret the error messages from many commands; in particular from *cc*, *grep -n* and *lint*). *Parse-errors* then does a *find-file* on the first such filename and a *goto-line* on its linenumber. When you have dealt with the error on that line (perhaps editing lines elsewhere in your program in the process) you can type ˆX ˆN (*next-error*) to move to the next error (perhaps in a different file). Or you can type ˆX ˆP (*previous-error*) or ESC X *current-error*. The rules JOVE uses to interpret error message in a buffer are specified in the variable *error-format-string*.

The action following the JOVE command *spell* is similar, except that it calls (automatically) the command *parse-spelling-errors-in-buffer* instead of *parse-errors*.

If you already have a file called *errs* containing, say, C compiler messages then you can get JOVE to interpret the messages by invoking it as:

% jove −p errs

12.2.1.2. Filtering

filter-region	Pass the region through a command and replace it with the output.

Suppose your buffer contains a table. You make this table the region (put mark at the start of it and point at the end). Type ESC X *filter-region sort*<return> (or any other command). Your table will be passed through the *sort* command and be replaced by the sorted version of itself. The old version is placed in the kill ring, and you can restore the status quo by obeying the *yank-pop* command.

13. Directory Handling

To save having to use absolute pathnames when you want to edit a nearby file JOVE maintains a *current directory* and allows you to move around the filesystem just as a shell would.

cd dir	Change to the specified directory.
pushd [dir]	Like *cd*, but saves the old directory on the directory stack. With no directory argument, simply exchanges the top two directories on the stack and *cd*s to the new top.
pushlibd	Does a *pushd* on the directory containing all of JOVE's standard customization files.
popd	Take the current directory off the stack and restore the previous one.
dirs	Display the contents of the directory stack.

The names and behavior of these commands were chosen to mimic those in the c-shell.

14. Major and Minor Modes

14.1. Major Modes

To help with editing particular types of file, say a document or a C program, JOVE has several *major modes*. Each mode defines rules as to which characters constitute a "word" (for the purposes of regular-expressions, language identifiers, abbreviations, and double-clicking in *xjove*), how indentation is to be performed, and maybe other specialized services. These are currently as follows:

14.1.1. Fundamental Mode

This is the simplest mode, with no frills, and is used when you are operating within the message line at the bottom of the screen (hence it is the mode of the Minibuf).

14.1.2. Text mode

This is the default major mode. Nothing special is done beyond making apostrophe (') be a word character.

14.1.3. C mode

In this mode, "$" and "_" are word characters, and there are special facilities for indentation. Using the *auto-execute-command* command, you can make JOVE enter *C Mode* whenever you edit a file whose name ends in *.c*.

14.1.3.1. Indentation Commands

To save having to lay out C programs "by hand", JOVE has an idea of the correct indentation of a line, based on the surrounding context. When you are in C Mode, JOVE treats tabs specially — typing a Tab at the beginning of a new line means "indent to the right place" (actually, it just goes back to the line containing the nearest unmatched "{", and indents 1 Tab more than that line). The indentation will be in multiples of the variable *c-indentation-increment* (which defaults to 8). Closing braces are also handled specially, and are indented to match the corresponding open brace.

If you Tab in the middle of a (...) (for example, you call a function whose actual-parameters stretch over many lines) then you have a choice depending on the variable *c-argument-indentation*. If its value is −1, you will be aligned with the corresponding actual-parameter on the line above. Otherwise,*c-ar gument-indentation* gives the extra number of characters by which to indent this continuation line.

If you really want a Tab to mean a single Tab on some particular occasion, you can always precede it by a ˆQ (*quoted-insert*).

14.1.3.2. Parenthesis and Brace Matching

	show-match-mode	Toggle the *Show Match* minor mode.
ESC ˆN	*forward-list*	Move forwards over a (...).
ESC ˆP	*backward-list*	Move backwards over a (...).
ESC ˆD	*down-list*	Move forward to just inside the next (...).
ESC ˆU	*backward-up-list*	Move backwards to the start of the enclosing (...).

To check that parentheses and braces match the way you think they do, turn on the *Show Match* minor mode (ESC X *show-match-mode*). Then, whenever you type a close brace or parenthesis, the cursor moves momentarily to the matching opener, if it is currently visible. If it's not visible, JOVE displays the line containing the matching opener on the message line.

If your parenthesized expressions are already typed, then you may find ESC ˆN and ESC ˆP useful to find a closing parenthesis to match an opening one somewhere just after point, or a closing one to match an opening one somewhere just before point. Note that these commands handle all kinds of parentheses ((...), [...] and {...}) and properly matched internal pairs are skipped over. These two commands take arguments and go in the opposite direction if the argument is negative. Likewise, the commands ESC D (*down-list*) and ESC U (*backward-up-list*) may be used to find more or less (respectively) deeply nested parentheses.

14.1.3.3. C Tags

Often when you are editing a C program, especially someone else's code, you see a function call and wonder what that function does. So you have to suspend the edit, *grep* for the function-name in every .c file that might contain it, and finally visit the appropriate file.

To avoid this diversion or the need to remember which function is defined in which file, many systems provide a program called *ctags(1)*, which takes a set of source files and looks for function definitions, producing a file called *tags* as its output.

ˆX T	*find-tag*	Find the file/line where the specified tag is declared.
	find-tag-at-point	Find the tag immediately following point.

JOVE has a command called ˆX T (*find-tag*) that prompts you for the name of a function (a *tag*), looks up the tag reference in the previously constructed *tags* file, then performs a *find-file* on the file containing that tag, with point positioned at the definition of the function. There is another version of this command, *find-tag-at-point*, that uses the identifier at point.

So, when you've added new functions to a module, or moved some old ones around, run the *ctags* program to regenerate the *tags* file. JOVE looks in the file specified by the variable *tag-file*. The default is "./tags", i.e. the tag file in the current directory. If you wish to use an alternate tag file you use ˆU ˆX T and JOVE will prompt for a file name.

To begin an editing session looking for a particular tag, use the *−t tag* command line option to JOVE. For example, say you wanted to look at the file containing the tag *SkipChar*, you would invoke JOVE as:

% jove −t SkipChar

14.1.4. Lisp mode

In this mode, any of the characters "!$%&∗+−/:<=>?ˆ_{|}˜" and Delete are word characters (in other words, "words" are Lisp atoms). The mode is analogous to*C Mode*, b ut performs the indentation needed to lay out Lisp programs properly.

14.1.4.1. Parenthesis Matching

In addition to the parenthesis matching commands available under C mode, we have:

ESC ˆF	*forward-s-expression*	Move backward over an atom or list.
ESC ˆB	*backward-s-expression*	Move forward over an atom or a list.
	grind-s-expression	Re-indent an s-expression.
ESC ˆK	*kill-s-expression*	Kill from point to the end of an s-expression.

In fact the first two of these commands work in other modes also, but for “atom“ read “identifier”.

14.2. Minor Modes

In addition to the major modes, JOVE has a set of minor modes whose state is controlled by the following commands:

auto-indent-mode
auto-fill-mode
dbx-mode
over-write-mode
read-only-mode
show-match-mode
word-abbrev-mode

With no argument, these commands toggle the mode. With a zero argument they turn it *off*, and with any other argument they turn it *on*.

14.2.1. Auto Indent

In this mode, JOVE indents each line the same way as that above it. That is, the Return key in this mode acts as the Linefeed key ordinarily does. This mode is only likely to be useful if you are afflicted with a keyboard without a Linefeed key.

14.2.2. Auto Fill

In *Auto Fill* mode, a newline is automatically inserted when the line length exceeds the right margin. This way, you can type a whole paragraph without having to use the Return key.

14.2.3. DBX

This mode is useful if dbx is being run in an interactive window. It is described in the section on interactive windows.

14.2.4. Over Write

In this mode, any text typed in will replace the previous contents (the default is for new text to be inserted and “push” the old along). This is useful for editing an already-formatted diagram in which you want to change some things without moving other things around on the screen.

14.2.5. Read Only

In this mode, modifying the buffer is inhibited. This mode is set automatically on any attempt to read a file for which you do not have write permission.

14.2.6. Show Match

Move the cursor momentarily to the matching opening parenthesis when a closing parenthesis is typed.

14.2.7. Word Abbrev

In this mode, every word you type is compared to a list of word abbreviations; whenever you type an abbreviation, it is replaced by the text that it abbreviates. This can save typing if a particular word or phrase must be entered many times. For example, your programming language might have reserved words that you customarily type in upper case (identifiers etc. being in lower case). So you might define B as an abbreviation for BEGIN, E for END, P for

PROCEDURE, and so on.

define-global-word-abbrev	Define a new global abbreviation.
define-word-abbrev	Define a new abbreviation within the current major mode.
edit-word-abbrev	Edit the list of abbreviations.
write-word-abbrev-file	Write the list of abbreviations to a file.
read-word-abbrev-file	Read a list of abbreviations from a file.

The abbreviations and their expansions are held in a list that looks like:

abbrev:phrase

for example

jove:jonathan’s own version of EMACS

Use *define-global-word-abbrev* to add an entry that is to be effective in all buffers and *define-word-abbrev* for an entry that is to be effective only in buffers of the same major mode as the selected buffer. Use*edit-wor d-abbrev* to edit the list (it enters a recursive edit on a buffer containing the list — use ˆX ˆC (*exit jove*) when you are finished). Use *write-word-abbrev-file* to write the list to a file and *read-word-abbrev-file* to read it back again (this command might be used in your **.joverc** file) .

If the variable *auto-case-abbrev* is *on,* and the abbreviations in the list are all in lower case (as in the “jove” example above) then, whenever you type “jove” you will get

jonathan’s own version of EMACS

but if you type “Jove” you will get

Jonathan’s own version of EMACS

and if you type “JOVE” (with at least 2 upper case letters) you will get

Jonathan’s Own Version Of EMACS

On the other hand, if the variable *auto-case-abbrev* is *off* (as it should be for the reserved word example) the case of the abbreviation is significant and must be matched exactly.

15. Macros

ˆX (	*begin-kbd-macro*	Start recording your commands.
ˆX)	*end-kbd-macro*	Stop recording your commands.
ESC I	*make-macro-interactive*	Call for a parameter to be typed at this point.
ˆX E	*execute-kbd-macro*	Replay the recording.
	name-kbd-macro	Name the recording.
	define-macro	Define a named macro.
	execute-macro	Execute a named macro.
	write-macros-to-file	Write all named macros to a file.
	bind-macro-to-key	Bind named macro to a key-sequence.
	bind-macro-to-word-abbrev	Bind named macro to an abbrev.

Although JOVE has many powerful commands, you often find that you have a task that no individual command can do. JOVE allows you to define your own commands from sequences of existing ones. The easiest way to do this is “by example”.

15.1. Keyboard Macros

First you type ˆX ((*begin-kbd-macro*). Next you “perform” the commands which will constitute the body of the macro (they are executed as well as being remembered). Then you type ˆX) (*end-kbd-macro*). You now have a *keyboard macro*.

To run this command sequence again, type ˆX E (*execute-keyboard-macro*).

If your macro needs a parameter (a filename to be opened, perhaps), include the command needing the parameter (e.g. ˆX ˆF) at the appropriate place in the macro followed immediately by ESC I (*make-macro-interactive*). When the macro is executed, you will be given an opportunity to type in the actual-parameter at this point.

15.2. Named Macros

You may give the keyboard macro a name using the *name-keyboard-macro* command (or you may create a named macro from scratch using the *define-macro* command). We're still not finished because all this hard work will be lost if you leave JOVE. What you do next is to save your macros into a file with the *write-macros-to-file* command. To retrieve your macros in the next editing session, you can simply execute the *source* command on that file, or include that file in your personal **.joverc** file.

A named macro can be executed by typing ESC X *execute-macro* <macro-name><return>. It is unfortunate that macro names are kept in a different name space than command names, so that you cannot type ESC X <macro-name>. This may well be changed in a future release.

15.3. Binding

Finally, if you find all this bothersome to type and re-type, there is a way to bind the macro to a key. The binding is made with the *bind-macro-to-key* command, or alternatively the *bind-macro-to-word-abbrev* command (in which case the macro will be executed upon typing the abbrev word you have specified — it will get expanded as well unless it was an abbreviation for nothing).

16. Customizing Jove

16.1. The jove.rc and .joverc files

JOVE is aware of a directory, the *sharedir*, in which system-wide customization files are kept. Chief amongst these is the file **jove.rc** which is read each time JOVE is started up (**jove.rc** may then initiate the reading of other files in the sharedir, such as initialization files for specific terminals). After that, JOVE reads your personal **.joverc** in your $HOME directory, if you have one. And if all that is not enough, you may at any time read other customization files using the *source* command.

The JOVE distribution comes with a recommended **jove.rc** file together with specific **jove.rc.TERM** files for various terminals. It is up to system administrators to decide whether to use these as they stand or to modify them to accord with local conventions. Note that these files are well commented and worthy of study by those who decide to "roll their own".

There are command-line options that can be used when JOVE is started to substitute a different *sharedir* or to suppress reading of the **jove.rc** or **.joverc** files or both — see the Man page for JOVE. Thus everything is ultimately under the control of the individual JOVE user.

16.2. The source Command

Type ESC X *source* <filename><return> to read and obey the commands in <filename>. If a numeric argument is supplied to *source*, it will silently ignore a request for a non-existent file (otherwise an error message will be produced). The format of the*sour ce*d file, as of the **jove.rc** and **.joverc** files, is as follows.

Each line consists of a command name (no need to precede it with ESC X) followed by whatever parameters that command requires. To give a numeric argument to the command, simply precede the command name by a number. Thus it is possible to to do anything that the user could do while JOVE is running.

But there is more than this. You can say

```
    if <shell-command>
        <command>
        <command>
    else
        <command>
        <command>
    endif
```

The <shell-command> is run, and if it succeeds the first lot of <command>s is obeyed, and otherwise the second lot (the *else* part is optional). Another variant of this feature allows you to say, in place of *if* <shell-command>, *ifenv* <environment-variable> <pattern>, which succeeds if <environment-variable> exists and if its value matches the regular expression <pattern> (an anchored match from the beginning of the variable value). There is a similar variant *ifvar* <jove-variable> <pattern>, which succeeds if the specified JOVE variable exists and if its value matches <pattern>. These conditional commands can be nested in the usual way; also indentation and empty lines have no effect. Finally, any line that starts with a "#" is treated as a comment and is ignored by JOVE.

Here are some examples taken from the provided **jove.rc**.

```
pushlibd

# This is for the shell window.  Supports sh, csh and ksh.
set process-prompt ^[^%$#]*[%$#]

# Modern terminals do not need ^S/^Q for flow control.
# The exceptions (e.g. vt100) should turn it off again in jove.rc.TERM.
set allow-^S-and-^Q on

# source any TERMinal-specific rc file
1 source jove.rc.$TERM

popd
```

The *pushlibd* ensures that any files it tries to read will be taken from the *sharedir* (observe the matching *popd* at the end). Then follow some settings of variables such as *process-prompt* (see the discussion of interactive processes earlier in this manual). Observe how environment variables such as $TERM are honored within parameters, and note how that *source* command was given a numeric argument so that there would be no complaint if the file **jove.rc.$TERM** did not exist.

On the other hand, if**jo ve.rc.$TERM** does exist for the particular terminal specified in $TERM, that file will now be *source*d. It will likely set many key bindings particular to that terminal, and then say

```
define-macro keychart ^[xpushlibd^M
        ^U^[Xshell-command-with-typeout cat keychart.$TERM^M
        ^[Xpopd^M
# except that should really have been all on one line
bind-macro-to-key keychart ^[[~
```

Quite some mouthful! What it does is to define a macro *keychart* (the hard way) and bind it to ESC [~. In general, any terminal for which extensive key bindings are provided ought to define this macro and bind it to a suitable key (preferably the one inscribed "Help"). When this key is pressed, it will cause the file **keychart.$TERM** to be displayed on the screen in *typeout* style. This file should exhibit a map of the terminal's keyboard, showing what has been bound to each key. The*shar edir* contains several such keychart files.

Although Control characters may be stored as themselves in these files (as produced by the *quoted-insert* command, for example), it is better to store them using an explicit "^" (e.g. as ^C), since this form is accepted by the *source* command, and editing files in this form is much easier.

16.3. Key Re-binding

Many of the commands built into JOVE are not bound to specific keys. You must type ESC X <command-name> (*execute-named-command*) in order to invoke these commands. Also, many of the keys to which commands *are* bound are hard to remember (although at least compatible across all terminals) whilst all sorts of interesting keys on the particular keyboard remain unused. For both these reasons, JOVE makes it possible to *bind* commands to keys.

bind-to-key	Bind a command to a key-sequence.
bind-macro-to-key	Bind a named macro to a key-sequence.
bind-macro-to-word-abbrev	Bind a named macro to an abbrev.
bind-keymap-to-key	Bind an extra key-sequence to a keymap.
describe-bindings	Exhibit all key bindings as a screen typeout.
local-bind-to-key	As bind-to-key, for use in the selected buffer only.
local-bind-macro-to-key	As bind-macro-to-key, for use in the selected buffer only.

local-bind-keymap-to-key	As bind-keymap-to-key, for use in the selected buffer only.
process-bind-to-key	Bind interactive process command to a key-sequence.
process-bind-macro-to-key	Bind a macro within interactive processes only.
process-bind-keymap-to-key	Bind a keymap within interactive processes only.

Although these commands can be typed in by the user, they are mostly intended for use in *source*d files. Here are some more examples from **jove.rc**.

```
# if you have job control, this puts Jove out of the way temporarily.
bind-to-key pause-jove ^[S
bind-to-key pause-jove ^[s

# The following apply to shell windows. Note the use of ^C^C, ^C^D etc.,
process-bind-to-key interrupt-process ^C^C
process-bind-to-key eof-process ^C^D

# This makes the arrow keys work on most terminals.
bind-to-key previous-line ^[[A
bind-to-key next-line ^[[B
```

When a command is *bound* to a key any future hits on that key will invoke that command. All the printing characters are initially bound to the command *self-insert*. Thus, typing any printing character causes it to be inserted into the text. To unbind a key, simply bind it to the fictitious command *unbound*.

Observe how key-sequences are often derived from common prefixes, such as ˆX ..., ESC ... (to be typed as ˆ[... in binding commands) and ESC [... (or ˆ[[...). Internally, JOVE creates tables for each prefix encountered, but it cannot create a new prefix from a manually entered *bind-* command (it does not know when you have finished your binding). To overcome this, give the *bind-* command an argument and terminate it with a Return (this applies automatically within *source*d files). Obviously, you must not have two bindings where one is a prefix of the other.

Very rarely, you may want two prefixes to be regarded as equivalent for all commands (for example, you have a keyboard with no ESC key, and it would be tedious to have to rebind every command in the system with a different prefix). In this case, you can type ESC X*bind-k eymap-to-key* <named-keymap> <prefix-key-sequence>. The only recognized <named-keymap>s are “ESC-map” and “Ctlx-map”, and the customary replacement for ESC is “‘”.

For historical reasons, the Escape key is often referred to as “Meta”. Indeed, if your terminal has a Meta-key which forces the 8th-bit of a character, and if the variable *meta-key* is *on*, you may type Y whilst holding the Meta-key down to achieve the same effect as when typing ESC Y.

16.3.1. The Provided Terminal Bindings

The terminals for which keybindings have been provided are a mixed bunch (we would welcome suggestions for other common terminals). However, there are certain principles which were followed in setting them up.

1. Groupings of keys that are found in bindings for other terminals were adhered to so far as possible. Rather than saying that the F1 key always does so-and-so, groupings of Function Keys that are physically associated on the keyboard were mapped onto similar groupings on other keyboards, even though the engravings on them might be quite different.

2 Keys which do related things should be close together.

3 Keys which customarily do certain things under other editors normally used with that keyboard should do the same (or similar) things under JOVE.

4 Keys which have suggestive engravings on them should do what the engravings suggest. Sometimes, this necessitated the creation of a macro where no JOVE command existed to do precisely that job (for example, the macro *kill-line*).

16.4. Auto-execution of Commands

It is useful, when a file is recognized as being in a particular programming language, for the appropriate major mode and other relevant facilities to be set up automatically in any buffer into which such a file is read.

auto-execute-command	Obey the given command for each filename matched by the given regular-expression.
auto-execute-macro	Obey the given macro likewise.

Here is an example taken from **jove.rc**.

```
    # set various language modes.
    1 auto-execute-command c-mode .∗.[chy]$
    1 auto-execute-command lisp-mode .∗.l$ .∗.lisp$ .∗.scm$
    # all the languages need show-match-mode.
    1 auto-execute-command show-match-mode .∗.[lchyfp]$ .∗.lisp$ .∗.scm$
```

The effect of this is that whenever a filename matches the regular-expression “.∗.[chy]$” the command*c-mode* is obeyed in the buffer into which the file is being read, and similarly for *lisp-mode*. An attempt is then made to set *show-match-mode* for both C and Lisp programs. Observe that all the *auto-execute-command*s in this example have an argument of 1. This argument is passed on to the obeyed command so that, for example, it is ensured that *c-mode* is definitely set to be *on*, rather than merely being toggled.

16.5. Customizing the Mode Line

The format of the mode line is controlled by the variable *mode-line*. Here is a suggested setting.

```
%[Jove%]%w%w%c(%M)%3c[%b:%n]%2c"%f"%2c%m∗-%m∗-%2c%p%2s%(%d%e(%t)%)
```

and here is what it all means.

%[...%]	Puts brackets around Jove when in a recursive edit.
%w%w	Warns with >> if the window is scrolled left.
(%M)	Gives the current major and minor modes.
[%b:%n]	Shows the buffer name and number.
"%f"	Shows the filename.
%m∗−%m∗−	Displays ∗∗ if the buffer is modified, −− if not.
%p	In process windows only, shows the status of the process.
%d	Shows the current directory.
(%t)	Shows the time of day.

Everything else is layout. See the full description of the *mode-line* variable for further details.

17. Xjove and Xterm

If you run JOVE on a workstation equipped with the X-Windows system from M.I.T., then it is advised to run JOVE under one of the terminal emulators *xjove* or *xterm*. *Xterm* is provided as a standard part of the X-Windows system, but the facilities provided are a small subset of those available with *xjove*, which was written especially to support JOVE. However, *xjove* suffers from the disadvantage that it must be compiled under the XView Toolkit which, although available free from M.I.T., may not be available on your system. Note that, in either case, special keybindings must be provided (see the files **jove.rc.sun-cmd** and **jove.rc.xterm**). See the Man page for*xjo ve* for how to call it and the flags and options available.

17.1. Basic Mouse Operations

When running under *xjove* or *xterm* you may click the LEFT mouse button in order to set the position of point, and the MIDDLE mouse button to set the position of point and mark. If you hold the MIDDLE button down, you may sweep it along, leaving mark where you started and point where you finished, thus defining the region. If you hold the Control key down while you are doing this, the region is copied to the kill ring, as with the *copy-region* command, and if you hold both the Control and Shift keys down, the region is killed (and sent to the kill ring) as with the *kill-region* command. To have the killed text yanked at some other point, click the LEFT mouse button there, holding the Control key down at the same time.

To switch to a different window, simply click either button in the window you want to be in (note that this does not affect point or mark in that window — it takes two clicks to change windows and then change point).

To scroll rapidly to a different part of the buffer, simply click either mouse button in the mode line at a position corresponding to the percentage way down the file you want to be. It helps to have set the variable *scroll-bar on* so that you can see by the uninverted part of the mode line which part of the buffer is currently visible in the window. When you have finished, the mouse pointer should be exactly in the middle of the uninverted region.

17.2. Additional Xjove Features

When running under *xjove* there are some additional facilities. Firstly, the setting when pointing into the mode line is more sensitive, because it notes the mouse position to the nearest pixel instead of the nearest character, and it is possible to hold the mouse button down and watch the window scrolling as you drag it (although this can consume considerable machine resources and it may have difficulty in keeping up). Also, it is possible to follow the position of point in real time as you drag the mouse around when delineating a region.

If you do a double click with the MIDDLE button in *xjove*, it will set the region spanning the word you were over (or spanning the gap if you were between words). Note that the definition of “word” here follows the major mode. If you do a triple click, it will likewise select the whole line. These operations may be combined with the Control key, or the Control and Shift keys together, to obtain copying and killing as before.

If your keyboard has keys marked Paste and Cut, it is possible to bring text from another part of the buffer (even from a different window) without changing the position of point (this is useful if you are constructing text at some point, bringing in fragments from other places — you would prefer not to have to keep moving point to those other places to acquire some text for the kill ring, only to have to move it back again before yanking). To do this, with point where you want the text to be inserted, you hold the Paste key down while you select a region with the MIDDLE button (multi-clicking or dragging as usual). When you have finished, a copy of the region you selected will appear at point, with point moved beyond it (since it went via the kill ring, this text is also available for conventional yanking subsequently). If you change your mind in mid stream, let go of Paste before releasing MIDDLE. Likewise, if you do the same thing holding the Cut key down, the text will also be killed from its original position.

Finally, if you press the RIGHT mouse button, you will be offered a menu which enables you to issue any JOVE command or set any JOVE variable. Particularly useful if you need some obscure and rarely used command, and cannot remember exactly what it is called.

18. Recovering from system/editor crashes

JOVE does not have an *Auto Save* mode, but it does provide a way to recover your work in the event of a system or editor crash. JOVE saves information about the files you are editing every *sync-frequency* changes to a special buffer, so as to make recovery possible. Since a relatively small amount of information is involved it is hardly even noticeable when JOVE does this. The variable *sync-frequency* says how often to save the necessary information, and the default is every 50 changes. 50 is a very reasonable number: if you are writing a paper you will not lose more than the last 50 characters you typed, which is less than the average length of a line.

If JOVE, or the operating system, crashes, you may now use the JOVE *recover* program to get back your files. You invoke this by running JOVE with the -r flag. See the Man page for JOVE for further details.

Another worthwhile precaution you can take is to set the variable *make-backup-files on*. Then, whenever you save a file with *save-file* it will leave behind the original version of that file with the name “#*filename*~”.

19. Alphabetical List of Commands and Variables

In this chapter, the standard binding is shown for each command which has one. Generally, these are the built-in bindings, but occasionally they are ones taken from the provided **jove.rc** file.

19.1. abort-char (variable)

This variable defines JOVE'S abort character. When JOVE reads this character from the keyboard, it stops what it is doing (unless the character is quoted in some way). Unfortunately, JOVE won't notice the character until it reads from the keyboard. The default value is ˆG. See also *interrupt-character*.

19.2. add-lisp-special (Not Bound)

This command is to tell JOVE what identifiers require special indentation in lisp mode. Lisp functions like *defun* and *let* are two of the default functions that get treated specially. This is just a kludge to define some of your own. It prompts for the function name.

19.3. allow-ˆS-and-ˆQ (variable)

This variable, when set, tells JOVE that your terminal will not need to use the characters ˆS and ˆQ for flow control, in which case JOVE will instruct the system's tty driver to pass them through as normal characters. Otherwise, if the tty driver was already using these characters for flow control, it will continue to do so. Certain terminals and communications systems require that this variable be set *off*; in other circumstances it is better set *on*.

19.4. allow-bad-characters-in-filenames (variable)

If set, this variable permits the creation of filenames which contain "bad" characters such as those from the set ∗&%!"‘[]{}. These files are harder to deal with, because the characters mean something to the shell. The default value is *off*.

19.5. append-region (Not Bound)

This appends the region to a specified file. If the file does not already exist it is created.

19.6. apropos (Not Bound)

This types out each command, variable and macro with the specified string in its name ("?" matches every name). For each command and macro that contains the string, the key sequence that can be used to execute the command or macro is printed; with variables, the current value is printed. So, to find all the commands that are related to windows, you type

: apropos window<Return> .

19.7. auto-case-abbrev (variable)

When this variable is *on* (the default), word abbreviations are adjusted for case automatically. If the abbreviation is typed with no uppercase letter, the expansion is not changed; if it is typed with one or more uppercase letters, the first character in the expansion is capitalized; additionally, if the abbreviation is typed with more than one uppercase letter, each letter in the expansion immediately preceded by whitespace or − is capitalized. For example, if "jove" were the abbreviation for "jonathan's own version of EMACS", the following table shows how the abbreviation would be expanded.

jove	jonathan's own version of EMACS
Jove	Jonathan's own version of EMACS
JOVE	Jonathan's Own Version Of EMACS
JoVe	Jonathan's Own Version Of EMACS

When this variable is *off*, upper and lower case are distinguished when looking for the abbreviation, i.e., in the example above, "JOVE" and "Jove" would not be expanded unless they were defined separately. See also the*wor d-abbrev-mode* command.

19.8. auto-execute-command (Not Bound)

This tells JOVE to execute a command automatically when a file whose name matches a specified pattern is read. The first argument is the command you wish to have executed. The second argument is the pattern, a regular expression that is matched against the start of the file name. If you wish to match a suffix, start the pattern with ".*"; to match every file, use that as the whole pattern. Any numeric argument will be passed on to the command when it is executed (this is useful when combined with commands that adjust a minor mode). For example, if you want to be in *show-match-mode* when you edit C source files (that is, files that end with **.c** or **.h**) you can type

```
: auto-execute-command show-match-mode .*\.[ch]$
```

Actually, this command toggles the Show Match minor mode, but since it is initially off, it will have the desired effect. For more certain control, give the *auto-execute-command* a non-zero numeric argument: this will be passed on to the *show-match-mode*.

19.9. auto-execute-macro (Not Bound)

This is like *auto-execute-command* except you use it to execute macros automatically instead of built-in commands.

19.10. auto-fill-mode (Not Bound)

This turns on or off the Auto Fill minor mode in the selected buffer. Without a numeric argument, the command toggles the mode; with a zero argument, the mode is turned off; with a non-zero argument, the mode is turned on. When JOVE is in Auto Fill mode it automatically breaks lines for you when you reach the right margin so you don't have to remember to hit Return. JOVE uses 78 as the right margin but you can change that by setting the variable *right-margin* to another value.

19.11. auto-indent-mode (Not Bound)

This turns on or off Auto Indent minor mode in the selected buffer. Without a numeric argument, the command toggles the mode; with a zero argument, the mode is turned off; with a non-zero argument, the mode is turned on. When JOVE is in Auto Indent mode, the *newline* command (which is normally bound to Return) acts identically to *newline-and-indent*: the new line is indented to the same position as the line you were just on. This is useful for lining up C code (or any other language (but what else is there besides C?)). Furthermore, if a line is broken because of Auto Fill mode, and Auto Indent mode is on, the new line will be indented as the old line was.

19.12. backward-character (^B)

This moves point backward over a single character or line-separator. Thus if point is at the beginning of the line it moves to the end of the previous line.

19.13. backward-list (ESC ^P)

This moves point backward over a list, which is any text between properly matching (...), [...] or {...}. It first searches backward for a ")" and then moves to the matching "(". This is useful when you are trying to find unmatched parentheses in a program. Arguments are accepted, and negative arguments search forwards. See also *backward-s-expression*.

19.14. backward-paragraph (Usually Not Bound)

This moves point backward to the beginning of the current or previous paragraph. Paragraphs are bounded by lines that match *paragraph-delimiter-pattern* (by default, those that are empty or look like troff or TeX commands). A change in indentation may also signal a break between paragraphs, except that JOVE allows the first line of a paragraph to be indented differently from the other lines. Arguments are accepted, and negative arguments search forwards.

19.15. backward-s-expression (ESC ^B)

This moves point backward over an s-expression, that is over a Lisp atom or a C identifier (depending on the major mode) ignoring punctuation and whitespace; or, if the nearest preceding significant character is one of ")]}", over a list as in *backward-list*. Arguments are accepted, and negative arguments search forwards.

19.16. backward-sentence (ESC A)

This moves point backward to the beginning of the current or previous sentence. JOVE considers the end of a sentence to be the characters ".", "!" or "?" followed by a Return or by one or more spaces. Arguments are accepted, and negative arguments search forwards.

19.17. backward-up-list (ESC ^U)

This is similar to *backward-list* except it backs up and OUT of the enclosing list. In other words, it moves backward to whichever of "([{" would match one of ")]}" if you were to type it right then. Arguments are accepted, and negative arguments search forwards as in *down-list*.

19.18. backward-word (ESC B)

This moves point backward to the beginning of the current or previous word. Arguments are accepted, and negative arguments search forwards.

19.19. bad-filename-extensions (variable)

This contains a list of words separated by spaces which are to be considered bad filename extensions, and so will not be included in filename completion. The default contains, amongst much else, **.o** so if you have **jove.c** and **jove.o** in the same directory, the filename completion will not complain of an ambiguity because it will ignore **jove.o**.

19.20. begin-kbd-macro (^X ()

This starts defining the keyboard macro by remembering all your key strokes until you execute *end-kbd-macro*, by typing "^X)". Because of a bug in JOVE you shouldn't terminate the macro by typing "ESC X end-kbd-macro"; *end-kbd-macro* must be bound to "^X)" in order to make things work correctly. Th*ee xecute-kbd-macro* command will execute the remembered key strokes. Sometimes you may want a macro to accept different input each time it runs. To see how to do this, see the *make-macro-interactive* command.

19.21. beginning-of-file (ESC <)

This moves point backward to the beginning of the buffer. This sometimes prints the "[Point pushed]" message to indicate that JOVE has set the mark so you can go back to where you were if you want. See also the variable *mark-threshold*.

19.22. beginning-of-line (^A)

This moves point to the beginning of the current line.

19.23. beginning-of-window (ESC ,)

This moves point to the beginning of the active window. If there is a numeric argument, point moves that many lines below the top line. With the default bindings, the sequence "ESC ," is the same as "ESC <" (*beginning-of-file*) but without the shift key on the "<", and can thus easily be remembered.

19.24. bind-keymap-to-key (Not Bound)

This is like *bind-to-key* except that you use it to attach a key sequence to a named keymap. The only reasonable use is to bind some extra key to *ESC-map* for keyboards that make typing ESC painful.

19.25. bind-macro-to-key (Not Bound)

This is like *bind-to-key* except you use it to attach a key sequence to a named macro.

19.26. bind-macro-to-word-abbrev (Not Bound)

This command allows you to bind a macro to a previously defined word abbreviation. Whenever you type the abbreviation, it will first be expanded as an abbreviation (which could be empty, of course), and then the macro will be executed. Note that if the macro moves point around, you should first *set-mark* and then *exchange-point-and-mark*.

19.27. bind-to-key (Not Bound)

This attaches a key sequence to an internal JOVE command so that future hits on that key sequence invoke that command. This is called a global binding, as compared to local bindings and process bindings. Any previous global binding of this key sequence is discarded. For example, to make “ˆW” erase the previous word, you type

```
: bind-to-key kill-previous-word ˆW .
```

It isn’t possible to have two globally bound key sequences where one is a prefix of the other: JOVE wouldn’t know whether to obey the shorter sequence or wait for the longer sequence. Normally, when the *bind-to-key* command is issued interactively, the key sequence is taken to end one keystroke after the longest sequence matching any proper prefix of another binding (thus no new prefix can be created). If the command is given a numeric argument, the key sequence is taken up to the next Return keystroke (kludge!); bindings to any prefix of the sequence are discarded. When the command is issued from a *source*d file, the key sequence is taken up to the end of the line (it is also processed so that control characters can and should be entered using the ˆA notation).

Note that neither process nor local bindings are changed by this command, although they can be eclipsed. Given a choice between bindings, the shortest is executed; if there is still a choice, a process binding is preferred to a local binding, and a local binding is preferred to a global binding.

19.28. buffer-position (Not Bound)

This displays the current file name, current line number, total number of lines, current character number, total number of characters, percentage of the way through the file, and the position of the cursor in the current line.

19.29. c-argument-indentation (variable)

This variable describes how to indent lines which are part of nested expressions in C. The default is −1, which means to indent a continued line by lining it up with the first argument of the current expression. Otherwise, the line will be indented by *c-argument-indentation* characters past the indent of the first line of the expression. For example, the default value produces:

```
Typeout(fmt, itoa(bcount++), line_cnt(b, nbuf),
        TypeNames[b->b_type],
        IsModified(b) ? "*" : b->b_ntbf ? "+" : NullStr,
        buf_width, b->b_name, filename(b));
```

19.30. c-indentation-increment (variable)

This defines a set of tabstops independent of the value of *tab-width*. This value will be used in C mode, and JOVE will insert the correct number of Spaces and Tabs to get the right behavior. For programmers that like to indent with 4 spaces, set this value to 4. Some people prefer to set this to 4 and leave tab-width set to 8. This will create files whose indentation steps in 4-space increments, and which look the same anywhere that tabs are expanded to 8 spaces (i.e. in most settings). Others prefer to have one tab character per indentation level, then fiddle the tab expansion width to get the appearance they like. They should set both *c-indentation-increment* and *tab-width* to 4. Whenever using a non-standard tab width (*tab-width*) you should only use tabs for indentation, and use spaces for all columnar alignment later in the lines.

19.31. c-mode (Not Bound)

This turns on the C major mode in the currently selected buffer. When in C or Lisp mode, Tab, “}”, and “)” behave a little differently from usual: They are indented to the “right” place for C (or Lisp) programs. In JOVE, the “right” place is simply the way the author likes it (but I’ve got good taste).

19.32. case-character-capitalize (Not Bound)

This capitalizes the character after point, i.e., the character under the cursor. If a negative argument is supplied that many characters before point are upper cased.

19.33. case-ignore-search (variable)

This variable, when *on*, tells JOVE to treat upper and lower case the same when searching. Thus “jove” would match “JOVE”, and “JoVe” would match either. The default value of this variable is *off*.

19.34. case-region-lower (Not Bound)

This changes all the upper case letters in the region to their lower case equivalents.

19.35. case-region-upper (Not Bound)

This changes all the lower case letters in the region to their upper case equivalents.

19.36. case-word-capitalize (ESC C)

This capitalizes the current word by making the current letter upper case and making the rest of the word lower case. Point is moved to the end of the word. If point is not positioned on a word it is first moved forward to the beginning of the next word. If a negative argument is supplied that many words before point are capitalized. This is useful for correcting the word just typed without having to move point to the beginning of the word yourself.

19.37. case-word-lower (ESC L)

This lower-cases the current word and leaves point at the end of it. If point is in the middle of a word the rest of the word is converted. If point is not in a word it is first moved forward to the beginning of the next word. If a negative argument is supplied that many words before point are converted to lower case. This is useful for correcting the word just typed without having to move point to the beginning of the word yourself.

19.38. case-word-upper (ESC U)

This upper-cases the current word and leaves point at the end of it. If point is in the middle of a word the rest of the word is converted. If point is not in a word it is first moved forward to the beginning of the next word. If a negative argument is supplied that many words before point are converted to upper case. This is useful for correcting the word just typed without having to move point to the beginning of the word yourself.

19.39. cd (Not Bound)

This changes the current directory.

19.40. character-to-octal-insert (Not Bound)

This inserts a Back-slash followed by the ascii value of the next character typed. For example, “ˆG” inserts the string “\007”.

19.41. clear-and-redraw (ESC ˆL)

This clears the entire screen and redraws all the windows. Use this when JOVE gets confused about what’s on the screen, or when the screen gets filled with garbage characters or output from another program.

19.42. comment-format (variable)

This variable tells JOVE how to format your comments when you run the command *fill-comment*. Its format is this:

```
<open pattern>%!<line header>%c<line trailer>%!<close pattern>
```

The %!, %c, and %! must appear in the format; everything else is optional. A newline (represented by %n) may appear in the open or close patterns. %% is the representation for %. The default comment format is for C comments. See *fill-comment* for more details.

19.43. compile-it (ˆX ˆE)

This compiles your program by running the command *make* into a buffer, and automatically parsing the error messages that are created (if any). See the*par se-errors* command. If *compile-it* is given a numeric argument, it will prompt for a command to run in place of the plain make and the command you enter will become the new default.

See also *error-format-string* which makes it possible to parse errors of a different format and see also the variable *error-window-size*.

19.44. continue-process (Not Bound)

This sends the signal SIGCONT to the interactive process in the current buffer, IF the process is currently stopped.

19.45. copy-region (ESC W)

This takes all the text in the region and copies it onto the kill ring buffer. This is just like running *kill-region* followed by the *yank* command. See the *kill-region* and *yank* commands.

19.46. current-error (Not Bound)

This moves to the current error in the list of parsed errors. See the *next-error* and *previous-error* commands for more detailed information.

19.47. date (Not Bound)

This prints the date on the message line.

19.48. dbx-format-string (variable)

This is the default regular-expression search string used by JOVE to parse output from *dbx* running in a shell process (see the *dbx-mode* command). You shouldn't have to change this unless you are using something other than *DBX*.

19.49. dbx-mode (Not Bound)

This turns on or off the DBX minor mode in the selected buffer. Without a numeric argument, the command toggles the mode; with a zero argument, the mode is turned off; with a non-zero argument, the mode is turned on. This mode only makes sense in a buffer running an interactive shell process. If you are running *dbx* in a window and and the buffer is in DBX minor mode, JOVE will automatically track the source location in another window. Whenever you type "where" or while you're stepping through a program, or when you reach a breakpoint, JOVE will present the source file in another window and move to the line that is being referenced. See also the variable *dbx-format-string*.

19.50. define-global-word-abbrev (Not Bound)

This defines a global abbreviation. See the*wor d-abbrev-mode* command.

19.51. define-macro (Not Bound)

This provides a different mechanism for defining keyboard macros. Instead of gathering keystrokes and storing them into the "keyboard-macro" (which is how *begin-kbd-macro* works), *define-macro* prompts for a macro name (terminated with Space, or Newline) and then for the actual macro body. If you wish to specify control characters in the macro, you may simply insert them (using the *quoted-insert* command) or by inserting the character 'ˆ' followed by the appropriate letter for that character (e.g., ˆA would be the two characters 'ˆ' followed by 'A'). You may use Back-slash to prevent the 'ˆ' from being interpreted as part of a control character when you really wish to insert one (e.g., a macro body "\ˆfoo" would insert the string "ˆfoo" into the buffer, whereas the body "ˆfoo" would be the same as typing ˆF and then inserting the string "oo"). See *write-macros-to-file* to see how to save macros.

19.52. define-mode-word-abbrev (Not Bound)

This defines a mode-specific abbreviation. See the*wor d-abbrev-mode* command.

19.53. delete-blank-lines (ˆX ˆO)

This deletes all the blank lines around point. This is useful when you previously opened many lines with the *newline-and-backup* command and now wish to delete the unused ones.

19.54. delete-buffer (ˆX K)

This deletes a buffer and frees up all the memory associated with it. Be careful(!) - once a buffer has been deleted it is gone forever. JOVE will ask you to confirm if you try to delete a buffer that needs saving. This command is useful for when JOVE runs out of space to store new buffers. See also the*er ase-buffer* command and the *kill-some-buffers* command.

19.55. delete-current-window (ˆX D))

This deletes the active window and moves point into one of the remaining ones. It is an error to try to delete the only remaining window.

19.56. delete-next-character (ˆD)

This deletes the character that's just after point (that is, the character under the cursor). If point is at the end of a line, the line-separator is deleted and the next line is joined with the current one. If an argument is given, that many characters are deleted and placed on the kill ring. If the argument is negative the deletion is forwards.

19.57. delete-other-windows (ˆX 1)

This deletes all the other windows except the current one. This can be thought of as going back into One Window mode.

19.58. delete-previous-character (DEL and ˆH)

This deletes the character that's just before point (that is, the character before the cursor). If point is at the beginning of the line, the line separator is deleted and that line is joined with the previous one. If an argument is given, that many characters are deleted and placed on the kill ring. If the argument is negative the deletion is backwards.

19.59. delete-white-space (ESC \)

This deletes all the Tabs and Spaces around point.

19.60. describe-bindings (Not Bound)

This types out a list containing each bound key and the command that gets invoked every time that key is typed. To make a wall chart of JOVE commands, set *send-typeout-to-buffer* to *on* and JOVE will store the key bindings in a buffer which you can save to a file and then print.

19.61. describe-command (ESC ?)

This waits for you to type a command and then prints an explanation of that command, together with its current bindings.

19.62. describe-key (ˆX ?)

This waits for you to type a key and then tells the name of the command that gets invoked every time that key is hit. Once you have the name of the command you can use the *describe-command* command to find out exactly what it does.

19.63. describe-variable (Not Bound)

This prints an explanation of a specified variable.

19.64. digit (ESC 0 through ESC 9)

Starts or continues the entry of a numeric argument with the digit typed. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified.

19.65. digit-0 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 0. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 0 key on the numeric keypad.

19.66. digit-1 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 1. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 1 key on the numeric keypad.

19.67. digit-2 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 2. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 2 key on the numeric keypad.

19.68. digit-3 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 3. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 3 key on the numeric keypad.

19.69. digit-4 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 4. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 4 key on the numeric keypad.

19.70. digit-5 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 5. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 5 key on the numeric keypad.

19.71. digit-6 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 6. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 6 key on the numeric keypad.

19.72. digit-7 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 7. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 7 key on the numeric keypad.

19.73. digit-8 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 8. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 8 key on the numeric keypad.

19.74. digit-9 (Not Bound)

Starts or continues the entry of a numeric argument with the digit 9. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the 9 key on the numeric keypad.

19.75. digit-minus (ESC −)

Starts the entry of a numeric argument with a minus sign. It continues reading digits until you type some other command. Then that command is executed with the numeric argument you specified. Sometimes it is handy to bind this to the − key on a numeric keypad. In the absence of further digits and unless otherwise stated (e.g. *next-page*), the argument −1 is assumed.

19.76. dirs (Not Bound)

This prints out the directory stack. See the *cd*, *pushd*, *pushlibd* and *popd* commands for more information.

19.77. disable-biff (variable)

When this is set, JOVE disables biff when you're editing and enables it again when you get out of JOVE, or when you pause to the parent shell or push to a new shell. (This means arrival of new mail will not be immediately apparent but will not cause indiscriminate writing on the display). The default is *off*, although it is always safe to set it *on*, even on systems that do not provide the biff facility. Note that the variable *mode-line* can be set up to announce the arrival of new mail during a JOVE session.

19.78. display-default-filenames (variable)

If this is set when JOVE asks for a filename, it will display the default (unless that would take too much of the prompt line).

19.79. display-filenames-with-bad-extensions (variable)

This variable affects only filename completion, in particular, what happens when "?" is typed while prompting for a file. When this variable is *on*, any files that end with one of the extensions defined by the variable *bad-filename-extensions* will be displayed with an "!" in front of their names. When *display-filenames-with-bad-extensions* is *off* the files will not be displayed at all. The default value is *on*.

19.80. down-list (ESC ˆD)

This is the opposite of *backward-up-list*. It enters the next list. In other words, it moves forward to whichever of "([{" it first encounters. Arguments are accepted, and negative arguments search backwards as in *backward-up-list*.

19.81. dstop-process (Proc: ˆC ˆY)

Send the signal SIGTSTP to the interactive process in the selected buffer when next it tries to read input. This is equivalent to sending the "dsusp" character (which most people have set to ˆY) to the process. This only works if you are in a buffer bound to an interactive process.

19.82. edit-word-abbrevs (Not Bound)

This creates (if necessary) a buffer with a list of each abbreviation and the phrase it expands into, and enters a recursive edit to let you change the abbreviations or add some more. The format of this list is "abbreviation:phrase" so if you add some more you should follow that format. It's probably simplest just to copy some already existing abbreviations and edit them. Use the *exit-jove* command to exit the recursive edit.

19.83. end-kbd-macro (ˆX))

This stops the definition of the keyboard macro. Because of a bug in JOVE, this must be bound to "ˆX)", or some key sequence which is one or two characters long. Anything else will not work properly. See*be gin-kbd-macro* for more details.

19.84. end-of-file (ESC >)

This moves point forward to the end of the buffer. This sometimes prints the "[Point pushed]" message to indicate that JOVE has set the mark so you can go back to where you were if you want. See also the variable *mark-threshold*.

19.85. end-of-line (ˆE)

This moves point to the end of the current line. If the line is too long to fit on the screen, it will be scrolled horizontally. This is described with the variables *scroll-width* and *scroll-all-lines*.

19.86. end-of-window (ESC .)

This moves point to the last character in the active window. If there is a numeric argument, the point moves that many lines above the bottom line. With the default bindings, the sequence “ESC .” is the same as “ESC >” (*end-of-file*) but without the shift key on the “>”, and can thus easily be remembered.

19.87. enhanced-keyboard (variable)

(IBM PC version only) This is a boolean variable which can be set to enable the enhanced AT-style keyboard. The enhanced keyboard contains function keys and key combinations that are not supported on the original IBM PCs and XTs. The default value is determined by a bit in the BIOS data area, but this method apparently does not work with a few BIOS implementations. WARNING: setting enhanced-keyboard *on* on systems without an enhanced keyboard will lock up your system and require you to reboot.

19.88. eof-process (ˆC ˆD)

Sends EOF to the current interactive process. This only works on versions of JOVE running under versions of with pty’s.

19.89. erase-buffer (Not Bound)

This erases the contents of the specified buffer. This is like *delete-buffer* except it only erases the contents of the buffer, not the buffer itself. If you try to erase a buffer that needs saving you will be asked to confirm it.

19.90. error-format-string (variable)

This is the error format string that is used by *parse-errors* to find the error messages in a buffer. The way it works is by using this string as a JOVE regular expression search string, where the \(...\) regular expression feature is used to pick out the file name and line number from the line containing an error message. For instance, a typical error message might look like this:

```
"file.c", line 540: missing semi-colon
```

For strings of this format, an appropriate value for *error-format-string* would be something like this:

```
^"\([^"]*\)", line \([0-9]*\):
```

What this means is, to find an error message, search for a line beginning with a double-quote. Then it says that all the following characters up to another double-quote should be remembered as one unit, namely the filename that the error is in (that is why the first set of parentheses is surrounding it). Then it says that after the filename there will be the string “, line ” followed by a line number, which should be remembered as a single unit (which is why the second set of parentheses is around that). The only constraint on the error messages is that the file name and line number appear on the same line. Most compilers seem to do this anyway, so this is not an unreasonable restriction.

If you do not know how to use regular expressions then this variable will be hard for you to use. Also note that you can look at the default value of this variable by printing it out, but it is a really complicated string because it is trying to accommodate the outputs of more than one compiler.

19.91. error-window-size (variable)

This is the percentage of the screen to use for the error-window on the screen. When you execute *compile-it* or *spell-buffer*, *error-window-size* percent of the screen will go to the error window. If the window already exists and is a different size, it is made to be this size. The default value is 20%.

19.92. exchange-point-and-mark (ˆX ˆX)

This moves point to mark and makes mark the old point. This is for quickly moving from one end of the region to the other.

19.93. execute-kbd-macro (ˆX E)

This executes the keyboard macro. If you supply a numeric argument the macro is executed that many times. See the *begin-kbd-macro* command for more details.

19.94. execute-macro (Not Bound)

This executes a specified macro. If you supply a numeric argument the macro is executed that many times.

19.95. execute-named-command (ESC X)

This is the way to execute a command that isn't bound to any key. When you are prompted with ": " you can type the name of the command. You don't have to type the entire name. After typing a few characters, Tab will fill in as many more as it can (as will Space, but that will also obey the command if it is now unambiguous). If you are not sure of the name of the command, type "?" and JOVE will print a list of all the commands that you could possibly match given what you've already typed. Once the command is unambiguous, typing Return will cause it to be obeyed.

If you don't have any idea what the command's name is but you know it has something to do with windows (for example), you can do "ESC X apropos window" and JOVE will print a list of all the commands that are related to windows. If you find yourself constantly executing the same commands this way you probably want to bind them to keys so that you can execute them more quickly. See the*bind-to-k ey* command.

19.96. exit-jove (ˆX ˆC)

This exits JOVE. If any buffers need saving JOVE will print a warning message and ask for confirmation. If you leave without saving your buffers all your work will be lost. If you made a mistake and really do want to exit then you can. If there are any interactive processes running, JOVE will also ask whether they should be terminated.

If you are in a recursive editing level *exit-jove* will return you from that. The selected buffer will be set back to the buffer that was current when the recursive edit was entered. Normally, point will be returned to its position at the time of entry, but if the *exit-jove* command is given a numeric argument, point is left at its most recent position within that buffer.

19.97. expand-environment-variables (variable)

When this variable is *on* JOVE will try to expand any strings of the form "$var" into the value of the environment variable "var" when asking for a filename. For example, if you type **$HOME/.joverc**, "$HOME" will be replaced with your home directory. The default value is *on*.

19.98. file-creation-mode (variable)

This variable has an octal value. It contains the mode (see*c hmod*(1)) with which files should be created. This mode gets modified by your current umask setting (see *umask*(1)). The default value is usually 0666 or 0644.

19.99. files-should-end-with-newline (variable)

This variable indicates that all files should always have a newline at the end. This is often necessary for line printers and the like. When set, if JOVE is writing a file whose last character is not a newline, it will add one automatically. The default value is *on*.

19.100. fill-comment (Not Bound)

This command fills in your C comments to make them pretty and readable. This filling is done according the variable *comment-format*.

```
/*
 * the default format makes comments like this.
 */
```

This can be changed by changing the *comment-format* variable. Other languages may be supported by changing the format variable appropriately. The formatter looks backwards from point for an open comment symbol. If found, all indentation is done relative to the position of the first character of the open symbol. If there is a matching close symbol, the entire comment is formatted. If not, the region between the open symbol and point is reformatted. The original text is saved in the kill ring; a *yank-pop* command will undo the formatting.

19.101. fill-paragraph (ESC J)

This rearranges words between lines so that all the lines in the current paragraph extend as close to the right margin as possible, ensuring that none of the lines will be greater than the right margin. The default value for *right-margin* is 78, but can be changed with the *set* and *right-margin-here* commands.

The rearrangement may cause an end of line to be replaced by whitespace. Normally, this whitespace is a single space character. If the variable *space-sentence-2* is *on*, and the end of the line was apparently the end of a sentence or the line ended with a colon, two spaces will be used. However, a sentence or colon followed by a single space already within a line will not be altered.

JOVE has a complicated algorithm for determining the beginning and end of the paragraph. In the normal case JOVE will give all the lines the same indent as they currently have, but if you wish to force a new indent you can supply a numeric argument to *fill-paragraph* and JOVE will indent each line to the column specified by the *left-margin* variable. See also the*left-mar gin* variable and *left-margin-here* command.

Filling a paragraph can do something that you didn't intend. For that reason the original text is saved on the kill ring and can be yanked back. Deleting the rubble is up to you.

19.102. fill-region (Not Bound)

This is like *fill-paragraph*, except it operates on a region instead of just a paragraph.

19.103. filter-region (Not Bound)

This sends the text in the region to a command, and replaces the region with the output from that command. For example, if you are lazy and don't like to take the time to write properly indented C code, you can put the region around your C file and *filter-region* it through *cb*, the C beautifier. If you have a file that contains a bunch of lines that need to be sorted you can do that from inside JOVE too, by filtering the region through the *sort* command. Before output from the command replaces the region JOVE stores the old text in the kill ring. If you are unhappy with the results a *yank-pop* command will get back the old text.

19.104. find-file (ˆX ˆF)

This reads a specified file into its own buffer and then selects that buffer. If you've already read this file into a buffer, that buffer is simply selected. If the file doesn't yet exist, JOVE will print "(New file)" so that you know. If possible, the buffer is named after the filename (ignoring any directory part).

19.105. find-tag (ˆX T)

This finds the file that contains the specified tag. JOVE looks up tags by default in the **tags** file in the current directory, as created by the command *ctags(1)*. You can change the default tag name by setting the *tag-file* variable to another name. If you specify a numeric argument to this command, you will be prompted for a tag file. This is a good way to specify another tag file without changing the default.

19.106. find-tag-at-point (Not Bound)

This finds the file that contains the tag that point is currently in. See *find-tag*.

19.107. first-non-blank (ESC M)

This moves point (backwards or forwards) to the indent of the current line.

19.108. forward-character (ˆF)

This moves point forward over a single character or line-separator. Thus if point is at the end of the line it moves to the beginning of the next one.

19.109. forward-list (ESC ˆN)

This moves point forward over a list, which is any text between properly matching (...), [...] or {...}. It first searches forward for a "(" and then moves to the matching ")". This is useful when you are trying to find unmatched parentheses in a program. Arguments are accepted, and negative arguments search backwards. See also*forwar d-s-expression*.

19.110. forward-paragraph (ESC])

This moves point forward to the end of the current or next paragraph. Paragraphs are bounded by lines that match *paragraph-delimiter-pattern* (by default, those that are empty or look like troff or TeX commands). A change in indentation may also signal a break between paragraphs, except that JOVE allows the first line of a paragraph to be indented differently from the other lines. Arguments are accepted, and negative arguments search backwards.

19.111. forward-s-expression (ESC ˆF)

This moves point forward over an s-expression, that is over a Lisp atom or a C identifier (depending on the major mode) ignoring punctuation and whitespace; or, if the nearest succeeding significant character is one of "([{", over a list as in *forward-list*. Arguments are accepted, and negative arguments search backwards.

19.112. forward-sentence (ESC E)

This moves point forward to the end of the current or next sentence. JOVE considers the end of a sentence to be the characters ".", "!" or "?", followed possibly by "’", "’’", or """, followed by a Return or whitespace. Arguments are accepted, and negative arguments search backwards.

19.113. forward-word (ESC F)

This moves point forward to the end of the current or next word.

19.114. fundamental-mode (Not Bound)

This sets the major mode to Fundamental. Fundamental mode is the mode of the Minibuf, and hence of anything typed in the message line.

19.115. gather-numeric-argument (ˆU)

This command is one of two ways to specify a numeric argument to a command. Typing this command once means, Do the next command 4 times. Typing it twice will do the next command 16 times, and so on. If at any point you type a number, then that number will be used instead of 4. For instance, ˆU 3 5 means do the next command 35 times (assuming *gather-numeric-argument* is bound to ˆU).

19.116. goto-line (ESC G)

If a positive numeric argument is supplied, point moves to the beginning of that line. If the argument is negative, it indicates how many lines from the end of the buffer to move point to. If no argument is supplied one is prompted for.

19.117. goto-window-with-buffer (Not Bound)

This command prompts for a buffer name and then selects that buffer. If the buffer is currently being displayed in one of the windows, that window is selected instead.

19.118. grind-s-expr (Not Bound)

When point is positioned on a "(", this re-indents that LISP expression.

19.119. grow-window (ˆX ˆ)

This makes the active window one line bigger. This only works when there is more than one window and provided there is room to change the size. See also *shrink-window*.

19.120. handle-tab (Tab)

This handles indenting to the “right” place in C and Lisp mode, and just inserts itself in Text mode.

19.121. highlight-attribute (variable)

(IBM PC version only) This specifies how the attribute (color) of a character is to be changed when it is highlighted. Highlighting is indicated by exclusive oring this value with the normal attribute for the character. The default is 16.

19.122. highlight-mark (variable)

When this is on, jove will highlight the mark if currently visible. The mark is highlighted with an underscore.

19.123. i-search-forward (Not Bound)

Incremental search. Like search-forward except that instead of prompting for a string and searching for that string all at once, it accepts the string one character at a time. After each character you type as part of the search string, it searches for the entire string so far. When you like what it found, type Return to finish the search. You can take back a character with DEL and the search will back up to the position before that character was typed. ˆG aborts the search.

19.124. i-search-reverse (Not Bound)

Incremental search. Like search-reverse except that instead of prompting for a string and searching for that string all at once, it accepts the string one character at a time. After each character you type as part of the search string, it searches for the entire string so far. When you like what it found, type Return to finish the search. You can take back a character with DEL and the search will back up to the position before that character was typed. ˆG aborts the search.

19.125. i-shell-command (Not Bound)

This is like *shell-command* except that it launches an interactive process and so lets you continue with your editing while the command is running. This is really useful for long running commands with sporadic output. See also the variable *wrap-process-lines*.

19.126. insert-file (ˆX ˆI)

This inserts a specified file into the selected buffer at point. Point is positioned at the beginning of the inserted file.

19.127. interrupt-character (variable)

This specifies what character should be used as the operating system’s tty driver interrupt character. When this character is typed, the tty driver generates SIGINT signal. This will interrupt a non-interactive process. If no such process is running, JOVE’S will offer you the option of continuing, or crashing JOVE’S (trying to save your work). This is a crude and desperate way to stop JOVE’S. Unfortunately there is no way to turn off the interrupt character. The default is ˆ]. See also *abort-char*.

19.128. insert-variable (Not Bound)

This inserts the value of the specified variable at the current point in the current buffer.

19.129. interrupt-process (Proc: ˆC ˆC)

This sends the signal SIGINT to the interactive process in the selected buffer. This only works if you are inside a buffer bound to an interactive process.

19.130. iproc-env-export (Not Bound)

This takes an argument of the form VARNAME=VALUE and replaces or sets VARNAME to that VALUE in the list of environment variables that are passed to interactive shell commands i.e. *shell* and *i-shell-command*.

19.131. iproc-env-show (Not Bound)

This shows (as temporary screen output or in a buffer, depending on the variable *send-typeout-to-buffer*) the environment that will be exported to interactive processes.

19.132. iproc-env-unset (Not Bound)

This takes an argument that is the name of an environment variable and removes it from the list of environment variables that will be passed to interactive processes.

19.133. jove-compiled-with (variable)

This is a special, read-only variable, containing the compiler, flags and options used to build Jove (on Unix-style machines)

19.134. jove-features (variable)

This is a special, read-only variable, set with a colon-delimited list of all the optional capabilities compiled into JOVE. The value will be something like
:unix:abbr:bak:biff:cmtfmt:fcomp:iproc:pty:lisp:proc:spell:rec:job:idchar:hl:tcap:ctype:
unix specifies a variety of traditional UNIX/POSIX capabilities (case-sensitive filenames, a Unix-style filesystem), as opposed to
msdos or *win32* which have case-insensitive filesystems, for example. The platform*ibmpcdos* indicates specialization for the IBM-PC platform, which in reality, is the only MS-DOS form remaining in existence (other non-IBM-PC platforms, like Heath/Zenith, DEC Rainbow, etc probably do not exist anymore, as is probably the case for the pre-OSX classic *mac* platform). Other tokens are
abbr (abbrev-mode exists),
bak (backup-files can be created),
biff (ability to disable screen-disruptions from old Unix-style mail notification),
cmtfmt (format C comments),
fcomp (filename completion),
iproc (interactive shell processes),
pipe (indicating that *iproc* is implemented using pipes and the *portsrv* helper program), or
pty (indicating that *iproc* is implemented using pseudo-terminals, available on any modern post-4.2BSD platform),
lisp (lisp mode is available),
proc (the ability to run non-interactive subshell processes with output to buffers, or switch entirely from JOVE to a child shell via *push-shell* and then return to JOVE when that child shell exits),
spell (the ability to run a spell checker on the contents of a buffer, parse and modify the checker output and then step through any misspelled words in the original buffer with ^X^N and ^X^P)
rec (periodically generate a recovery file so that one can restore unsaved changes from JOVE editors after a machine or program crash)
job (BSD job control is supported, so one can suspend JOVE with the pause-jove command and return to the shell that JOVE was started from, and later resume JOVE using a shell command like *fg*),
jsmall (JOVE was built with smaller limits on the number of lines it can support, probably only 32000)
idchar (JOVE will attempt to optimize screen display using insert/delete character modes and terminal control sequences), *hl* (enable highlight-mark capabiltiy to display a small underscore for the position of the mark, and enable the scroll-bar capability in the buffer mode line),
jtc (uses builtin ANSI/VT1xx/XTERM driver rather than *termcap/terminfo/curses*),
tcap (termcap database code is enabled)
tinfo (uses SystemV-style terminfo rather than BSD-style termcap calls),
ctype (uses the system ctype character classification rather than JOVE builtin capability)
iso88591 (uses builtin JOVE tables for ISO-8859-1 character classification rather than the system ctype capability).

19.135. jove-linked-with (variable)

This is a special, read-only variable, containing the linker, flags and libraries used to build Jove (on Unix-style machines)

19.136. kill-next-word (ESC D)

This kills the text from point to the end of the current or next word. The killed text is sent to the kill ring.

19.137. kill-previous-word (ESC DEL)

This kills the text from point to the beginning of the current or previous word. The killed text is sent to the kill ring.

19.138. kill-process (Not Bound)

This command prompts for a buffer name or buffer number (just as *select-buffer* does) and then sends the process in that buffer the signal SIGKILL.

19.139. kill-region (ˆW)

This deletes the text in the region and saves it on the kill ring. Commands that delete text but save it on the kill ring all have the word “kill” in their names. Use the *yank* command to get back the most recent kill.

19.140. kill-s-expression (ESC ˆK)

This kills the text from point to the end of the current or next s-expression. The killed text is sent to the kill ring.

19.141. kill-some-buffers (Not Bound)

This goes through all the existing buffers and asks whether or not to delete each one. If you decide to delete a buffer, and it turns out that the buffer is modified, JOVE will offer to save it first. This is useful for when JOVE runs out of memory to store lines (this only happens on PDP-11’s) and you have lots of buffers that you are no longer using. See also the*delete-b uffer* command.

19.142. kill-to-beginning-of-sentence (ˆX DEL)

This kills from point to the beginning of the current or previous sentence. If a negative numeric argument is supplied it kills from point to the end of the current or next sentence. The killed text is sent to the kill ring.

19.143. kill-to-end-of-line (ˆK)

This kills from point to the end of the current line. When point is at the end of the line (discounting any white space) the line-separator is also deleted and the next line is joined with current one. If a numeric argument is supplied that many lines are killed; if the argument is negative that many lines before point are killed; if the argument is zero the text from point to the beginning of the line is killed. The killed text is sent to the kill ring.

19.144. kill-to-end-of-sentence (ESC K)

This kills from point to the end of the current or next sentence. If a negative numeric argument is supplied it kills from point to the beginning of the current or previous sentence. The killed text is sent to the kill ring.

19.145. lc-ctype (variable)

This string variable determines how non-ASCII characters are displayed, and which characters are to be considered as upper-case, lower-case, printable, etc. The default is the implementation-defined native environment; under POSIX, it is determined by whichever of the environment variables LC_ALL, LC_CTYPE or LANG is first found to be set, and is otherwise "C". Some useful values of *lc-ctype* might be:

""	Default: the native environment.
"C"	Strict ASCII. All other characters greater than \177 rendered in octal.
"iso_8859_1"	Latin-1 alphabet.

19.146. left-margin (variable)

This is how far lines should be indented when Auto Indent mode is on, or when the *newline-and-indent* command is run (usually by typing Linefeed). It is also used by *fill-paragraph* and Auto Fill mode. If the value is zero (the default) then the left margin is determined from the surrounding lines.

19.147. left-margin-here (Not Bound)

This sets the *left-margin* variable to the current position of point. This is an easy way to say, “Make the left margin begin here,” without having to count the number of spaces over it actually is.

19.148. lib-dir-pathname (variable)

This tells JOVE where to find the machine-dependent helper programs: the *recover* program, run when *jove -r* is run after a crash to try to recover unsaved files, and on some older machines without pseudo-terminal support, the *portsrv* program, which handles multiplexed communication with the keyboard and interactive shells (if JOVE is compiled with SUBSHELL and PIPEPROCS configured)

19.149. lisp-mode (Not Bound)

This turns on the Lisp major mode. In Lisp mode, the characters Tab and “)” are treated specially, similar to the way they are treated in C mode. Also, Auto Indent mode is affected, and handled specially. See also the*c-mode* command.

19.150. list-buffers (ˆX ˆB)

This types out a list containing various information about each buffer. The list looks like this:

```
(* means the buffer needs saving)
NO    Lines  Type         Name            File
--    -----  ----         ----            ----
1     1      File         Main            [No file]
2     1      Scratch    * Minibuf         [No file]
3     519    File       * commands.doc    commands.doc
```

The first column lists the buffer’s number. When JOVE prompts for a buffer name you can either type in the full name, or you can simply type the buffer’s number. The second column is the number of lines in the buffer. The third says what type of buffer. There are four types: File, Scratch, Process and I-Process. “File” is simply a buffer that holds a file; “Scratch” is for buffers that JOVE uses internally; “Process” is one that holds the output from a command; “I-Process” is one that has an interactive process attached to it. The next column contains the name of the buffer. And the last column is the name of the file that’s attached to the buffer. In this case, both Minibuf and commands.doc have been changed but not yet saved. In fact Minibuf won’t be saved since it’s a Scratch buffer.

19.151. list-processes (Not Bound)

This makes a list somewhat like “list-buffers” does, except its list consists of the current interactive processes. The list looks like this:

```
Buffer          Status          Pid        Command
------          ------          ---        -------
*shell*         Running         18415      shell
fgrep           Done            18512      fgrep -n Buffer *.c
```

The first column has the name of the buffer to which the process is attached. The second has the status of the process; if a process has exited normally the status is “Done” as in fgrep; if the process exited with an error the status is “Exit N” where N is the value of the exit code; if the process was killed by some signal the status is the name of the signal that was used; otherwise the process is running. The last column is the name of the command that is being run.

19.152. local-bind-keymap-to-key (Not Bound)

This is like *local-bind-to-key* except that you use it to attach a key sequence to a named keymap. The only reasonable use is to bind some extra key to *ESC-map* for keyboards that make typing ESC painful.

19.153. local-bind-macro-to-key (Not Bound)

This is like *local-bind-to-key* except you use it to attach a key sequence to a named macro.

19.154. local-bind-to-key (Not Bound)

This is like *bind-to-key*, except that the binding is only enabled when the selected buffer is the buffer that was current when the command was executed. In other words, the binding only applies to the selected buffer.

19.155. macify (variable)

(Mac version only) When this variable is on, JOVE will use the standard Macintosh file-selector dialog in place of the traditional JOVE Minibuffer.

19.156. mail-check-frequency (variable)

This is how often (in seconds) JOVE should check your mailbox for incoming mail. If you set this to zero JOVE won't check for new mail. See also the *mode-line*, *mailbox* and *disable-biff* variables. The default is 60.

19.157. mailbox (variable)

Set this to the full pathname of your mailbox. JOVE will look here to decide whether or not you have any unread mail. This defaults to **/usr/spool/mail/$USER**, where "$USER" is set to your login name.

19.158. make-backup-files (variable)

If this variable is set, then whenever JOVE writes out a file, it will move the previous version of the file (if there was one) to "#filename˜". This is often convenient if you save a file by accident. The default value of this variable is *off*.

19.159. make-buffer-unmodified (ESC ˜)

This makes JOVE think the selected buffer hasn't been changed even if it has. Use this when you accidentally change the buffer but don't want it considered changed. Watch the mode line to see the ∗ disappear when you use this command.

19.160. make-macro-interactive (ESC I)

This command is meaningful only while you are defining a keyboard macro, and when you are expecting input in the message line. Ordinarily, when a command in a macro definition requires a trailing text argument (file name, search string, etc.), the argument you supply becomes part of the macro definition. If you want to be able to supply a different argument each time the macro is used, then while you are defining it, you should give the *make-macro-interactive* command just before typing the argument which will be used during the definition process. Note: you must bind this command to a key in order to use it; you can't say "ESC X make-macro-interactive".

19.161. mark-threshold (variable)

This variable contains the number of lines point may move by before the mark is set. If, in a search or some other command that may move point, point moves by more than this many lines, the mark is set so that you may return easily. The default value of this variable is 22 (one screenful, on most terminals). See also the commands *search-forward*, *search-reverse*, *beginning-of-file* and *end-of-file*.

19.162. match-regular-expressions (variable)

When set, JOVE will match regular expressions in search patterns. This makes special the characters ., ∗, [and]. See the JOVE Manual for a full discussion of regular-expressions.

19.163. meta-key (variable)

You should set this variable to *on* if your terminal has a real Meta key which forces the 8th bit of each character. If your terminal has such a key, then a key sequence like ESC Y can be entered by holding down Meta and typing Y. On the IBM PC, this variable affects how ALT is interpreted. On the Macintosh, it affects how Option is interpreted. NOTE: In some older systems, JOVE must switch the tty to raw mode to accept the 8-bit characters generated by a meta key. Unfortunately, the *interrupt-character* does not generate an interrupt in raw mode.

19.164. mode-line (variable)

The format of the mode line can be determined by setting this variable. The items in the line are specified using a format similar to that used by *printf(3)*, with the special things being marked as "%x". Digits may be used between the '%' and the 'x' to mean repeat that many times. 'x' may be:

C	checks for new mail, and displays "[New mail]" if there is any (see also the *mail-check-frequency* and *mailbox* variables)
F	the current file name, with leading path stripped
M	the current list of major and minor modes
b	the selected buffer name
c	the fill character (-)
d	the current directory
e	extra space in mode line is distributed evenly among the places %e is used (used for justifying, separating, or centering parts of the mode line)
f	the current file name
ixy	x, when the buffer's file has been changed behind JOVE's back, y, when not
mxy	x, when the buffer is modified or y, when not
n	the selected buffer number
p	interactive process status for process windows
s	space, but only if previous character is not a space
t	the current time (updated automatically)
w	a '>' for windows which are scrolled left
[ ]	the square brackets printed when in a recursive edit
( )	items enclosed in %(... %) will only be printed on the bottom mode line, rather than copied when the window is split

In addition, any other character is simply copied into the mode line. Characters may be escaped with a backslash. To get a feel for all this, try typing "ESC X print mode-line" and compare the result with your current mode line.

19.165. mode-line-attribute (variable)

(IBM PC version only) This specifies the screen attribute (color) for characters in the mode line. The default is 112 (black on white).

19.166. mode-line-should-standout (variable)

If set, the mode line will be printed in reverse video, if your terminal supports it. The default for this variable is *on*.

19.167. name-kbd-macro (Not Bound)

This copies the keyboard macro and gives it a name freeing up the keyboard macro so you can define some more. Keyboard macros with their own names can be bound to keys just like built in commands can. See the *define-macro*, *source* and *write-macros-to-file* commands.

19.168. newline (Return)

This divides the current line at point moving all the text to the right of point down onto the newly created line. Point moves down to the beginning of the new line. In Auto Indent mode, the new line will be indented to match the old line.

19.169. newline-and-backup (ˆO)

This divides the current line at point moving all the text to the right of point down onto the newly created line. The difference between this and *newline* is that point does not move down to the beginning of the new line.

19.170. newline-and-indent (Linefeed)

This behaves in any mode the same way as *newline* does in Auto Indent mode.

19.171. next-error (ˆX ˆN)

This moves to the next error in the list of errors that were parsed with *parse-errors*. In one window the list of errors is shown with the current one always at the top. If the file that contains the error is not already in a buffer, it is read in. Its buffer is displayed in another window and point is positioned in this window on the line where the error occurred.

19.172. next-line (ˆN)

This moves point down to the corresponding position on the next line (or the end of that line if it does not extend so far).

19.173. next-page (ˆV)

This displays the next page of the selected buffer by taking the bottom line of the window and redrawing the window with it at the top. If there isn't another page in the buffer JOVE rings the bell. If a numeric argument of only − (with no digits) is supplied, the previous page is displayed. Otherwise, if a numeric argument is supplied the screen is scrolled up that many lines, exactly as in the *scroll-up* command; if the argument is negative the screen is scrolled down.

19.174. next-window (ˆX N)

This moves into the next window. Windows live in a circular list so when you're in the bottom window and you try to move to the next one you are moved to the top window. It is an error to use this command with only one window.

19.175. number-lines-in-window (Not Bound)

This displays the line numbers for each line in the buffer being displayed. The number isn't actually part of the text; it's just printed before the actual buffer line is. To turn this off you run the command again; it toggles.

19.176. one-key-confirmation (variable)

If this variable is set, a single keystroke of y or n is expected in answer to yes/no questions. Normally, a yes/no question must be answered with any non-empty prefix of yes or no, followed by a Return

19.177. over-write-mode (Not Bound)

This turns Over Write minor mode on in the selected buffer. Without a numeric argument, the command toggles the mode; with a zero argument, the mode is turned off; with a non-zero argument, the mode is turned on. When on, this mode changes the way the self-inserting characters work. Instead of inserting themselves and pushing the rest of the line over to the right, they replace or over-write the existing character. Also, DEL replaces the character before point with a space instead of deleting it. When Over Write mode is on "OvrWt" is displayed in the mode line.

19.178. page-next-window (ESC ˆV)

This displays the next page in the next window. It switches to the next window, performs a *next-page* command (with any numeric argument), and switches back to the original window. Note that an argument of just "−" will thus display the previous page.

19.179. paren-flash () }])

This command causes the characters bound to it to be inserted, and then to partake in C mode curly brace indentation, Lisp mode parenthesis indentation, and the Show Match mode paren/curly-brace/square-bracket flashing.

19.180. paragraph-delimiter-pattern (variable)

When JOVE is searching for a paragraph boundary, if this pattern (a regular expression) matches the start of a line, that line is treated as a paragraph delimiter. The default pattern recognizes blank lines, troff control lines, and lines starting with a TeX control sequence.

There is a special provision for TeX: if a line is matched by the pattern, and the match is of exactly an initial \, that line is only treated as a delimiter if the next line also starts with \.

19.181. paren-flash-delay (variable)

How long, in tenths of a second, JOVE should pause on a matching parenthesis in Show Match mode. The default is 5.

19.182. parse-errors (Not Bound)

This takes the list of C compilation errors (or the output from another program in an acceptable format) in the selected buffer and parses them for use with the *next-error*, *previous-error* and *current-error* commands. This is a very useful tool and helps with compiling C programs or, when used in conjunction with the *grep* command, with making changes to a bunch of files. JOVE finds each file that has an error and remembers each line that contains an error. It doesn't matter if later you insert or delete some lines in the buffers containing errors; JOVE remembers where they are regardless. *current-error* is automatically executed after one of the parse commands, so you end up at the first error. The variable *error-format-string* specifies, by means of regular-expressions, the format of errors to be recognized. Its default value can handle messages from *cc*, *cpp*, *lint* and *grep −n*.

19.183. parse-spelling-errors-in-buffer (Not Bound)

This parses a list of words in the selected buffer and looks them up in another buffer that you specify. It is invoked automatically by the *spell-buffer* command.

19.184. pause-jove (ESC S)

This stops JOVE and returns control to the parent shell. This only works on systems that have the job control facility. To return to JOVE you type “fg” to the shell.

19.185. pop-mark (Not Bound)

JOVE remembers the last eight marks and you use *pop-mark* to go backward through the ring of marks. If you execute *pop-mark* enough times you will eventually get back to where you started. This command is also executed when you run *set-mark* with a numeric argument.

19.186. popd (Not Bound)

This pops one entry off the directory stack. Entries are pushed with the *pushd* or *pushlibd* commands. The names were stolen from the C-shell and the behavior is the same.

19.187. previous-error (ˆX ˆP)

This is the same as *next-error* except it goes to the previous error. See*ne xt-error* for documentation.

19.188. previous-line (ˆP)

This moves point up to the corresponding position on the previous line (or the end of that line if it does not extend so far).

19.189. previous-page (ESC V)

This displays the previous page of the selected buffer by taking the top line and redrawing the window with it at the bottom. If a numeric argument of only − (with no digits) is supplied, the next page is displayed. Otherwise, if a numeric argument is supplied the screen is scrolled down that many lines, exactly as in the *scroll-down* command; if the argument is negative the screen is scrolled up.

19.190. previous-window (ˆX P or ˆX O)

This moves into the previous window. Windows live in a circular list so when you're in the top window and you try to move to the previous one you are moved to the bottom window. It is an error to use this command with only one window.

19.191. print (Not Bound)

This displays the value of a JOVE variable in the message line.

19.192. proc-env-export (Not Bound)

This takes an argument of the form VARNAME=VALUE and replaces or sets VARNAME to that VALUE in the list of environment variables that are passed to subshell commands i.e. *shell-command*, *compile-it*, *spell-buffer*, etc.

19.193. proc-env-show (Not Bound)

This shows (as temporary screen output or in a buffer, depending on the variable *send-typeout-to-buffer*) the environment that will be exported to subshell commands.

19.194. proc-env-unset (Not Bound)

This takes an argument that is the name of an environment variable and removes it from the list of environment variables that will be passed to subshell commands.

19.195. process-bind-keymap-to-key (Not Bound)

This is like *process-bind-to-key* except that you use it to attach a key sequence to named keymap. The only reasonable use is to bind some extra key to *ESC-map* for keyboards that make typing ESC painful.

19.196. process-bind-macro-to-key (Not Bound)

This is like *process-bind-to-key* except you use it to attach a key sequence to a named macro.

19.197. process-bind-to-key (Not Bound)

This command is identical to *bind-to-key*, except that it only affects your bindings when you are in a buffer attached to an interactive process. When you enter the process buffer, any keys bound with this command will automatically take their new values. When you switch to a non-process buffer, the old bindings for those keys will be restored. For example, you might want to execute

```
process-bind-to-key stop-process ˆC ˆZ
process-bind-to-key interrupt-process ˆC ˆC
```

Then, when you start up an interactive process and switch into that buffer, ˆC ˆZ will execute *stop-process* and ˆC ˆC will execute *interrupt-process*. Bindings effective only in process windows are shown with a "Proc:" prefix in this manual and by the *apropos* and *describe-bindings* commands.

19.198. process-newline (Proc: Return)

This command is normally bound to Return as if by a *process-bind-to-key* so that it will only be bound in a process window. JOVE does two different things depending on where you are when you hit Return. When you're in the last line of the interactive process buffer, point moves to the end of the line, the line is terminated, and the line is made available as input to the process. When point is positioned in some other line, that line is copied to the end of the buffer (with the prompt stripped) and point is moved there with it, so you can then edit that line before sending it to the process. This command must be bound to the key you usually use to enter shell commands (Return), or else you won't be able to enter any. See the variable *process-prompt*.

19.199. process-prompt (variable)

What a prompt looks like from the *shell* and *i-shell-command* processes. The default is "% ", the default C-shell prompt. This is actually a regular expression search string. So you can set it to be more than one thing at once using the \| operator. For instance, for LISP hackers, the prompt can be

```
"% \|-> \|<[0-9]>: ".
```

19.200. process-send-data-no-return (Not Bound)

This is like *process-newline* except it sends everything to the process without the newline. Normally, when you type return in a process buffer it sends everything you typed including the Return. This command just provides a way to send data to the process without having to send a newline as well.

19.201. push-shell (Not Bound)

This spawns a child shell and relinquishes control to it. Within this shell, $1 can be used to refer to the filename (if any) of the selected buffer. This works on any version of but this isn't as good as *pause-jove* because it takes time to start up the new shell and you get a brand new environment every time. To return to JOVE, simply exit the shell.

19.202. pushd (Not Bound)

This pushes a directory onto the directory stack and cd's into it. It asks for the directory name but if you don't specify one it switches the top two entries on the stack. It purposely behaves the same as C-shell's *pushd*.

19.203. pushlibd (Not Bound)

Performs same function as *pushd* except that it pushes the Jove sharable library directory. This directory holds the system-wide **jove.rc** and the text used by the *describe-command* and *describe-variable* commands. It is mainly intended for use with the **jove.rc** file.

19.204. pwd (Not Bound)

This prints the pathname of the working directory, as in the *pwd* command.

19.205. query-replace-string (ESC Q)

This replaces strings matching a specified regular-expression with a specified replacement string. When a match is found, point is moved to it and then JOVE asks what to do. The options are:

Space or Y or y	to replace this match and go on to the next one.
Period	to replace this match and then stop.
DEL, BS, or N or n	to skip this match and go on to the next one.
ˆR or R or r	to enter a recursive edit. This lets you temporarily suspend the replace, do some editing, and then return to continue where you left off. To continue with the *query-replace-string*, use the *exit-jove* command.
ˆW	to delete the match and then enter a recursive edit.
ˆU or U or u	to undo all changes to the last modified line and continue the search from the start of that line.
! or P or p	to go ahead and replace the remaining matches without asking, as in *replace-string*.
Return or Q or q	to stop the *query-replace-string*.
ˆL	to redraw the screen

It is often useful to include a piece of the matched string in the replacement, especially if the piece was not matched by literal text. To select which part of the matched string is to be used, the corresponding part of the pattern is bracketed with \(and \). More than one set of brackets may be used, as long as they are properly nested. The matching substring is selected in the replacement string using \ followed by a digit: \1 for the first, \2 for the second, and so on. Conveniently, \0 always stands for the complete matched string, as if the whole regular expression were bracketed. For example, the following command will reverse pairs of comma-separated numbers:

```
: query-replace-string \([0-9]*\),\([0-9]*\) with \2,\1
```

The search for a match starts at point and goes to the end of the buffer, so to replace in the entire buffer you must first go to the beginning. Each subsequent search starts at the position after the previous match; if the previous match was an empty string, the search is first advanced one character to prevent unbounded repetition.

19.206. quit-process (Proc: ˆC ˆ\)

Send the signal SIGQUIT to the interactive process in the selected buffer. This is equivalent to sending the "quit" character (which most people have bound to ˆ\) to the process. This only works if you are in a buffer bound to an interactive process.

19.207. quoted-insert (ˆQ or ˆˆ)

This lets you insert characters that normally would be executed as other JOVE commands. For example, to insert "ˆF" you type "ˆQ ˆF" (assuming*quoted-insert* is bound to ˆQ). NUL cannot be represented in the b uffer, so *quoted-insert* will insert "ˆ@" in its stead. On the IBM PC under DOS, non-ASCII keystrokes are seen by JOVE as a hex FF character followed by another character; *quoted-insert* will quote both characters.

19.208. read-only-mode (Not Bound)

This turns on or off the Read-only minor mode. Without a numeric argument, the command toggles the mode; with a zero argument, the mode is turned off; with a non-zero argument, the mode is turned on. When a buffer is in Read-only mode, any attempt to modify the buffer will fail. When a file is found, and it's not got write permission, JOVE automatically puts the buffer in read-only mode. This is very helpful when you are in environments which use source control programs like RCS and SCCS. It prevents accidents like making a bunch of changes and only THEN discovering that you haven't checked the file out for making changes.

19.209. read-word-abbrev-file (Not Bound)

This reads a specified file that contains a bunch of abbreviation definitions, and makes those abbreviations available. See the *word-abbrev-mode* command.

19.210. recursive-edit (Not Bound)

This enters a recursive editing level. This isn't really very useful. I don't know why it's available for public use. I think I'll delete it some day.

19.211. redraw-display (ˆL)

This vertically centers the line containing point within the window. If that line is already in place, the screen is first cleared and then redrawn. If a numeric argument is supplied, the line is positioned at that offset from the top of the window. For example, "ESC 0 ˆL" positions the line containing point at the top of the window (assuming *redraw-display* is bound to ˆL).

19.212. rename-buffer (Not Bound)

This lets you rename the selected buffer.

19.213. replace-in-region (Not Bound)

This is the same as *replace-string* except that it is restricted to occurrences between point and the mark.

19.214. replace-string (ESC R)

This replaces all occurrences of a specified string with a specified replacement string. This is just like *query-replace-string* except that it replaces without asking.

19.215. right-margin (variable)

Where the right margin is for Auto Fill mode and the *fill-paragraph* and *fill-region* commands. The default is 78.

19.216. right-margin-here (Not Bound)

This sets the *right-margin* variable to the current position of point. This is an easy way to say, “Make the right margin begin here,” without having to count the number of spaces over it actually is.

19.217. save-file (ˆX ˆS or ˆX S or ˆX \)

This saves the selected buffer to the associated file. This makes your changes permanent so you should be sure you really want to do it. If the buffer has not been modified *save-file* refuses to do the save. If you really do want to write the file you must use *write-file*.

19.218. save-on-exit (variable)

If this is *on* when JOVE is about to exit, it will ask, for each modified buffer, whether you wish it to be saved. See *write-modified-files*.

19.219. scroll-all-lines (variable)

When this is *off*, (the default) horizontal scrolling will only affect the line containing point. When it is *on*, horizontal scrolling will affect the whole window. See also the*scr oll-width* variable.

19.220. scroll-bar (variable)

When this is turned *on*, a section of the mode line at the foot of each window is left in not-reverse-video, to show the position of the window relative to the whole of the file represented by that buffer (however, if the whole of the buffer is within the window, the whole mode line remains inverted).

19.221. scroll-down (ESC Z)

This scrolls the screen one line down. If the line containing point moves past the bottom of the window, point is moved up to the top of the window. If a numeric argument is supplied that many lines are scrolled; if the argument is negative the screen is scrolled up instead. See the *previous-page* command.

19.222. scroll-left (Not Bound)

This scrolls the text in the active window to the left. If a numeric argument is specified then the text is scrolled that number of columns. Otherwise, the text is scrolled by the number of columns specified by the variable *scroll-width*. If the variable *scroll-all-lines* is ON then *scroll-left* may actually do nothing if the scrolling would cause point not to be visible. A negative argument scrolls right. If the *mode-line* variable is suitably set, an indication that the text is scrolled will be given in the mode line.

19.223. scroll-right (Not Bound)

This scrolls the text in the active window to the right. If a numeric argument is specified then the text is scrolled that number of columns. Otherwise, the text is scrolled by the number of columns specified by the variable *scroll-width*. If the variable *scroll-all-lines* is ON then *scroll-right* may actually do nothing if the scrolling would cause point not to be visible. A negative argument scrolls left.

19.224. scroll-step (variable)

How many lines should be scrolled if the *previous-line* or *next-line* commands move you off the top or bottom of the screen. You may wish to decrease this variable if you are on a slow terminal. The default value is 0, which means to center the current line in the window. If the value is negative, the behavior is slightly different. If you move off the top of the window, and *scroll-step* is, say, −5 then the new line will be displayed 5 lines from the bottom of the window. If you move off the bottom of the window, the new line will be positioned 5 lines from the top of the window.

19.225. scroll-up (ˆZ)

This scrolls the screen one line up. If the line containing point moves past the top of the window, point is moved down to the top of the window. If a numeric argument is supplied that many lines are scrolled; if the argument is negative the screen is scrolled down instead. See also the *next-page* command.

19.226. scroll-width (variable)

Just as a buffer may be too long to be completely displayed in a window, a line may be too wide. JOVE handles wide lines through horizontal scrolling, displaying only a portion of the line. This variable affects horizontal scrolling. If point is outside the displayed portion of its line, but is within the specified number of columns beyond either side, the line is scrolled that much. Otherwise, the line will be scrolled to center point. The default value is 10. If the variable is 0, centering will always be used. See also the *scroll-all-lines* variable.

19.227. search-exit-char (variable)

Set this to the character you want to use to exit incremental search. The default is Newline, which makes *i-search* commands compatible with normal string search.

19.228. search-forward (ˆS or ˆ\)

This searches forward for a specified search string and positions point at the end of the string if it's found. If the string is not found point remains unchanged. This searches from point to the end of the buffer, so any matches before point will be missed. If point is moved by more than the variable *mark-threshold*, the old point will be pushed.

19.229. search-forward-nd (Not Bound)

This is just like *search-forward* except that it doesn't assume a default search string, and it doesn't set the default search string. This is useful for defining macros, when you want to search for something, but you don't want it to affect the current default search string.

19.230. search-reverse (ˆR)

This searches backward for a specified search string and positions point at the beginning if the string if it's found. If the string is not found point remains unchanged. This searches from point to the beginning of the buffer, so any matches after point will be missed. If point is moved by more than the variable *mark-threshold*, the old point will be pushed.

19.231. search-reverse-nd (Not Bound)

This is just like *search-reverse* except that it doesn't assume a default search string, and it doesn't set the default search string. This is useful for defining macros, when you want to search for something, but you don't want it to affect the current default search string.

19.232. select-buffer (ˆX B)

This selects a new or already existing buffer making it the current one. You can type either the buffer name or number. If you type in the name you need only type the name until it is unambiguous, at which point typing Tab or Space will complete it for you. If you want to create a new buffer you can type Return instead of Space, and a new empty buffer will be created.

19.233. select-buffer-1 (Not Bound)

This selects buffer number 1, if it exists.

19.234. select-buffer-10 (Not Bound)

This selects buffer number 10, if it exists.

19.235. select-buffer-2 (Not Bound)

This selects buffer number 2, if it exists.

19.236. select-buffer-3 (Not Bound)

This selects buffer number 3, if it exists.

19.237. select-buffer-4 (Not Bound)

This selects buffer number 4, if it exists.

19.238. select-buffer-5 (Not Bound)

This selects buffer number 5, if it exists.

19.239. select-buffer-6 (Not Bound)

This selects buffer number 6, if it exists.

19.240. select-buffer-7 (Not Bound)

This selects buffer number 7, if it exists.

19.241. select-buffer-8 (Not Bound)

This selects buffer number 8, if it exists.

19.242. select-buffer-9 (Not Bound)

This selects buffer number 9, if it exists.

19.243. self-insert (Most Printing Characters)

This inserts the character that invoked it into the buffer at point. Initially all but a few of the printing characters are bound to *self-insert*. See also*par en-flash*.

19.244. send-typeout-to-buffer (variable)

When this is *on* JOVE will send output that normally overwrites the screen (temporarily) to a buffer instead. This affects commands like *list-buffers*, *list-processes*, *shell-command-with-typeout*, and commands that use completion. The default value is *off*.

19.245. set (Not Bound)

This sets a specified variable to a new value.

19.246. set-mark (^@)

This sets the mark at the current position in the buffer. It prints the message “[Point pushed]” on the message line. It says that instead of “[Mark set]” because when you set the mark the previous mark is still remembered on a ring of eight marks. So “[Point pushed]” means point is pushed onto the ring of marks and becomes the value of “the mark”. To go through the ring of marks, use the *pop-mark* command. If you type this enough times you will get back to where you started. If a *set-mark* command is given a numeric argument, it acts like a *pop-mark* command.

19.247. share-dir-pathname (variable)

This tells JOVE where to find the machine-independent configuration files that it uses, typically some system-wide *jove.rc* configuration files read on startup, the formatted help information for the *describe-** commands and the tutorial. There is usually a compiled-in path that should be correct if Jove was properly installed. This path can also be overridden with the *-s* command-line option.

19.248. shell (variable)

The shell to be used with all the shell-∗ commands command. If your SHELL environment variable is set, it is used as the default value of *shell*; otherwise "/bin/csh" is the default. See also the description of the*shell-fla gs* variable to see how to change the flags passed to this shell.

19.249. shell (Not Bound)

This starts up an interactive shell in a window; if there is already an interactive shell, it just selects that buffer. JOVE uses "∗shell-n∗" (where**n** is the ar gument of the command) as the name of the buffer in which the interacting takes place. Thus different argument values refer to different interactive shells. See the JOVE manual for information on how to use interactive processes. See also the variable *wrap-process-lines*.

19.250. shell-command (ˆX !)

This runs a command and places the output from that command in a buffer. Within the command, $1 can be used to refer the the filename (if any) of the selected buffer. JOVE creates a buffer that matches the name of the command you specify and then attaches that buffer to a window. So, when you have only one window running, this command will cause JOVE to split the window and attach the new buffer to that window. Otherwise, JOVE finds the most convenient of the available windows and uses that one instead. If the buffer already exists it is first emptied (unless a numeric argument is specified). If it's already holding a file, not some output from a previous command, JOVE asks permission before emptying the buffer. Beware that if you go ahead, not only do you lose any unsaved changes that you made to the buffer, but the buffer's file name remains set, making it easy to later accidentally overwrite the original file. See also the variable *wrap-process-lines*.

19.251. shell-command-no-buffer (Not Bound)

This is just like *shell-command* except it just runs the command without saving the output to any buffer. It will report the success of the command in the usual way.

19.252. shell-command-to-buffer (Not Bound)

This is just like *shell-command* except it lets you specify the buffer to use.

19.253. shell-command-with-typeout (Not Bound)

This is just like *shell-command* except that instead of saving the output to a buffer, and displaying it in a window, this just types out the output in the same way that *list-buffers* does. Actually, how this behaves depends on the value of the variable *send-typeout-to-buffer*. If it is*on* then *shell-command-with-typeout* will beha ve just like *shell-command*. If a numeric argument is given, the "completed successfully" message at the end is suppressed.

19.254. shell-flags (variable)

This specifies a flag argument that directs the shell to take the next argument as a command to be executed. The default is "−c" (suitable for all known shells). Under MSDOS, the default is "/c" (suitable for command.com and similar MSDOS shells). Other MSDOS shells, such as MKS KSH require that this be changed to "−c". Under MSDOS, JOVE puts quotes around the command argument if *shell-flags* starts with "−". See the *shell* variable to change the default shell.

19.255. shift-region-left (Not Bound)

This shifts the region left by *c-indentation-increment* OR by the numeric argument, if one is supplied. If a negative argument is supplied the region is shifted the other way.

19.256. shift-region-right (Not Bound)

This shifts the region right by *c-indentation-increment* OR by the numeric argument, if one is supplied. If a negative argument is supplied the region is shifted the other way.

19.257. show-match-mode (Not Bound)

This turns on or off the Show Match minor mode in the selected buffer. Without a numeric argument, the command toggles the mode; with a zero argument, the mode is turned off; with a non-zero argument, the mode is turned on. This mode changes "}", ")" and "]" so that when they are typed they are inserted as usual, and then the cursor flashes back to the matching "{", "(" or "[" (depending on what was typed) for about half a second, and then goes back to just after the "}", ")" or "]" that invoked the command. This is useful for typing in complicated expressions in a program. You can change how long the cursor sits on the matching parenthesis by setting the *paren-flash-delay* variable in tenths of a second. If the matching "{", "(" or "[" isn't visible, the line containing the match is displayed on the message line.

19.258. shrink-window (Not Bound)

This makes the active window one line shorter, if possible. Windows must be at least 2 lines high, one for the text and the other for the mode line. See also *grow-window*.

19.259. source (Not Bound)

This reads a bunch of JOVE commands from a file. If a numeric argument is supplied to the *source* command, it will quietly do nothing if it cannot read the file.

The format of the file is the same as that in the **jove.rc** file, or your private **.joverc** in your home directory. There should be one command per line and it should be as though you were responding to an *execute-named-command* command while in JOVE. A command can be optionally preceded by a numeric argument. Lines commencing with a # are treated as comments. Control characters such as ˆR may be represented as themselves, or as "ˆ" followed by "R". ESC should be represented by ˆ[.

Sometimes it is useful to do different things in different circumstances. To make this possible, there are three conditional commands: *if*, *ifenv*, and *ifvar*. The*if* command tak es as an operand a shell command, which it runs. If the command succeeds, the commands after the *if*, until a line containing *else* or *endif*, are performed. Otherwise, these commands are suppressed and the commands after any *else*, up until an *endif*, are executed. Conditionals nest in the normal way. The*ifen v* command takes as operands the name of an environment variable and a pattern. If the environment variable is defined and its value matches the pattern, the *ifenv* succeeds. Note that patterns are anchored matches from the left. The *ifvar* command takes as operands the name of a JOVE variable and a pattern. If the JOVE variable is defined and its value matches the pattern, the *ifvar* succeeds.

For example, here are some lines from the file **jove.rc**.

```
bind-to-key pause-jove ˆ[S
bind-to-key pause-jove ˆ[s
set process-prompt ˆ[ˆ%$#]*[%$#]
# source any TERMinal-specific rc file
1 source jove.rc.$TERM
ifvar jove-features .*:iproc:
  process-bind-to-key interrupt-process ˆCˆC
endif
```

What they do is to provide two alternative key bindings for *pause-jove*, set the variable *process-prompt*, and attempt to call the *source* command on the file **jove.rc.$TERM**. Because of the numeric argument 1, there will be no complaint if this file cannot be found. If JOVE was compiled with interactive-processes enabled (in which case the string *:iproc:* will be part of the read-only JOVE variable *jove-features*), the command binds the key sequence Control-C Control-C to the *interrupt-process* command, which sends the SIGINT signal to interactive shell.

19.260. space-sentence-2 (variable)

If set *on*, two spaces are left after each sentence by commands such as *fill-paragraph*; otherwise, one space is left. The default is *on*.

19.261. spell-command-format (variable)

This is the format specification string for a command which will run a spell checker on a specified file and produce a list of misspelled words on the output. This variable (and related *spell-buffer* capability) is only available if Jove is built with support for subshells. The string must contain one instance of the specifier %s which will be replaced with the pathname of the file to be checked.

19.262. spell-buffer (Not Bound)

This saves the current buffer and runs it through the *spell* command (specified by the variable *spell-command-format*) and places the output in buffer "Spell". Then JOVE lets you edit the list of words, expecting you to delete the ones that you don't care about, i.e., the ones you know are spelled correctly. Then the*par se-spelling-errors-in-buffer* command comes along and finds all the misspelled words and sets things up so the error commands *next-error*, *previous-error* and *current-error* work. See also the variable *error-window-size*.

19.263. split-current-window (^X 2)

This splits the active window into two equal parts (providing the resulting windows would be big enough) and displays the selected buffer in both windows. Use*delete-other -windows* to go back to 1 window mode. If a numeric argument is supplied, the window is split "evenly" that many times (when possible).

19.264. start-remembering (Not Bound)

This is just another name for the *begin-kbd-macro* command. It is included for backward compatibility.

19.265. stop-process (Not Bound)

Send the signal SIGTSTP to the interactive process in the selected buffer. This is equivalent to sending the "stop" character (which most people have bound to ^Z) to the process. This only works if you are in a buffer bound to an interactive process.

19.266. stop-remembering (Not Bound)

This is just another name for the *end-kbd-macro* command. It is included for backward compatibility.

19.267. string-length (Not Bound)

This prints the number of characters in the string that point sits in. Strings are surrounded by double quotes. JOVE knows that "\007" is considered a single character, namely "^G", and also knows about other common ones, like "\r" (Return) and "\n" (Linefeed). This is mostly useful only for C programmers.

19.268. suspend-jove (Not Bound)

This is a synonym for *pause-jove*.

19.269. sync-frequency (variable)

The temporary files used by JOVE are forced out to disk every *sync-frequency* modifications. The default is 50, which really makes good sense. Unless your system is very unstable, you probably shouldn't fool with this.

19.270. tab-width (variable)

When JOVE displays a Tab character, it moves point forward to the next multiple of this variable. If the value is 0, tab is displayed as ^I, not whitespace. The default value is 8.

19.271. tag-file (variable)

This is the name of the file in which JOVE should look up tag definitions. The default value is "./tags".

19.272. text-attribute (variable)

(IBM PC version only) This specifies the screen attribute (color) for normal text characters. The default is 7 (white on black).

19.273. teach-jove (Not Bound)

This loads the Jove tutorial from from the **˜/teach-jove** file if it was already saved there, else it loads the reference copy from **SHAREDIR/teach-jove** into a buffer that will save to **˜/teach-jove,** replacing the old standalone *teachjove* command. The *-T* option, if given to JOVE on the command line, will start JOVE up with the tutorial loaded.

19.274. text-mode (Not Bound)

This sets the major mode to Text. This affects what JOVE considers as characters that make up words. For instance, Single-quote is not part of a word in Fundamental mode, but is in Text mode.

19.275. tmp-file-pathname (variable)

This tells JOVE where to put the tmp files, which is where JOVE stores buffers internally. The default is in **/tmp**, or as set up when your system was compiled, but if you want to store them somewhere else, you can set this variable. If your system crashes a lot it might be a good idea to set this variable to somewhere other than **/tmp** because the system removes all the files in **/tmp** upon reboot, and so you would not be able to recover editor buffers using the *jove -r* command.

NOTE: In order for this to work correctly you must set this variable BEFORE JOVE creates the tmp file. You can set this in your **.joverc** (the closer to the beginning the better), or as soon as you start up JOVE before you read any files.

19.276. transpose-characters (ˆT)

This switches the character before point with the one after point, and then moves forward one. This doesn't work at the beginning of the line, and at the end of the line it switches the two characters before point. Since point is moved forward, so that the character that was before point is still before point, you can use *transpose-characters* to drag a character down the length of a line.

19.277. transpose-lines (ˆX ˆT)

This switches the current line with the one above it, and then moves down one so that the line that was above point is still above point. This, like *transpose-characters*, can be used to drag a line down a page.

19.278. unbound (Not Bound)

This command acts as if an unbound key sequence were typed. In fact, that is its use: if you wish to unbind a key sequence, simply bind it to this command.

19.279. update-time-frequency (variable)

How often the mode line is updated (and thus the time). The default is 30 seconds.

19.280. use-i/d-char (variable)

If your terminal has insert/delete character capability you can tell JOVE not to use it by setting this to *off*. In my opinion it is only worth using insert/delete character at low baud rates. WARNING: if you set this to *on* when your terminal doesn't have insert/delete character capability, you will get weird (perhaps fatal) results.

19.281. version (Not Bound)

Displays the version number of this JOVE.

19.282. visible-bell (variable)

If the terminal has a visible bell, use it instead of beeping.

19.283. visible-spaces-in-window (Not Bound)

This displays an underscore character instead of each Space in the window and displays a greater-than followed by spaces for each Tab in the window. The actual text in the buffer is not changed; only the screen display is affected.

To turn this off you run the command again; it toggles.

19.284. visit-file (ˆX ˆV or ˆX ˆR)

This reads a specified file into the selected buffer replacing the old text. If the buffer needs saving JOVE will offer to save it for you. Sometimes you use this to start over, say if you make lots of changes and then change your mind. If that's the case you don't want JOVE to save your buffer and you answer "NO" to the question.

19.285. window-find (ˆX 4)

This lets you select another buffer in another window three different ways. This waits for another character which can be one of the following:

T	Finds a tag in the other window.
ˆT	Finds the tag at point in the other window
F	Finds a file in the other window.
B	Selects a buffer in the other window.

This is just a convenient short hand for *split-current-window* (or *previous-window* if there are already two windows) followed by the appropriate sequence for invoking each command. With this, though, there isn't the extra overhead of having to redisplay. In addition, you don't have to decide whether to use *split-current-window* or *previous-window* since *window-find* does the right thing.

19.286. word-abbrev-mode (Not Bound)

This turns on or off Word Abbrev minor mode in the selected buffer. Without a numeric argument, the command toggles the mode; with a zero argument, the mode is turned off; with a non-zero argument, the mode is turned on. Word Abbrev mode lets you specify a word (an abbreviation) and a phrase with which JOVE should substitute the abbreviation. You can use this to define words to expand into long phrases, e.g., "jove" can expand into "Jonathan's Own Version of Emacs"; another common use is defining words that you often misspell in the same way, e.g., "thier" => "their" or "teh" => "the". See the information on the*auto-case-abbr ev* variable.

There are two kinds of abbreviations: mode specific and global. If you define a Mode specific abbreviation in C mode, it will expand only in buffers that are in C mode. This is so you can have the same abbreviation expand to different things depending on your context. Global abbreviations expand regardless of the major mode of the buffer. The way it works is this: JOVE looks first in the mode specific table, and then in the global table. Whichever it finds it in first is the one that's used in the expansion. If it doesn't find the word it is left untouched. JOVE tries to expand words when you type a punctuation character or Space or Return. If you are in Auto Fill mode the expansion will be filled as if you typed it yourself.

19.287. wrap-process-lines (variable)

If this variable is *on*, the process output that is captured in a buffer is wrapped just before the line would have as many characters as there are columns on the screen. This introduces extra newlines, but it makes the output more readable. Note that the folding does not take into account that some characters (notably tabs) occupy more than one column of the display. The output of the*filter -region* command is not processed in this way because the extra newlines are presumed to be undesired in this case.

19.288. wrap-search (variable)

If set, searches will "wrap around" the ends of the buffer instead of stopping at the bottom or top. The default is *off*.

19.289. write-file (ˆX ˆW)

This saves the selected buffer to a specified file, and then makes that file the default file name for this buffer. If you specify a file that already exists you are asked to confirm over-writing it.

19.290. write-files-on-make (variable)

When set, all modified files will be written out before calling make when the *compile-it* command is executed. The default is *on*.

19.291. write-macros-to-file (Not Bound)

This writes the currently defined macros to a specified file in a format appropriate for reading them back in with the *source* command. The purpose of this command is to allow you to define macros once and use them in other instances of JOVE. See also the*define-macr o* command.

19.292. write-modified-files (ˆX ˆM)

This saves all the buffers that need saving. If you supply a numeric argument it asks, for each buffer, whether you really want to save it.

19.293. write-region (Not Bound)

This writes the text in the region to a specified file. If the file already exists you are asked to confirm over-writing it.

19.294. write-word-abbrev-file (Not Bound)

This writes the currently defined abbreviations to a specified file. They can be read back in and automatically defined with *read-word-abbrev-file*.

19.295. xj-mouse-commands (ˆX m∗)

Programs such as XJove and JoveTool generate these commands whenever a mouse button is pressed or released, or the mouse is moved while the button is pressed. They are followed by parameters giving parameters for the button pressed, the coordinates of the mouse, etc. They are not intended for direct use by the normal user.

The individual commands will now be described.

19.296. xj-mouse-copy-cut (ˆX m8)

Performs a *copy-region* if the CTRL key was down, or a *kill-region* if both CTRL and SHIFT were down. This command is normally bound to the release of button 2.

19.297. xj-mouse-line (ˆX m7)

Sets the region to be the whole line containing the cursor. This command is normally bound to a triple down click of button 2, and the presumed effects of the preceding double click are first undone.

19.298. xj-mouse-mark (ˆX m5)

Both point and mark are set to the cursor. This command is normally bound to the pressing of button 2.

19.299. xj-mouse-point (ˆX m[01249])

Point is set to the cursor. This command is normally bound to the single, double, and triple down-click and the dragging of button 1; also the dragging of button 2.

19.300. xj-mouse-word (ˆX m6)

Sets the region to be the word (or the gap between two words) containing the cursor. This command is normally bound to a double down click of button 2, and the presumed effects of the preceding single click are first undone.

19.301. xj-mouse-yank (ˆX m3)

Performs a *yank* if the CTRL key was down. This command is normally bound to the release of button 1.

19.302. xt-mouse (variable)

When set, JOVE sends XTerm escape sequences to enable and disable the mouse messages at appropriate times. Warning: due to the way XTerm encodes mouse events, if *meta-key* is set, mouse actions beyond column 95 or row 95 will be misunderstood; in any case, mouse actions beyond column 223 or row 223 will be misunderstood.

19.303. xt-mouse-commands (ESC [M∗)

Programs such as XTerm generate these commands whenever a mouse button is pressed or released. XTerm does not give the user as much power as XJove. They are followed by parameters specifying the button pressed, the coordinates of the mouse, etc. They are not intended for direct use by the normal user. Set the variable *xt-mouse* on to enable XTerm mouse mode.

The individual commands will now be described.

19.304. xt-mouse-mark (ˆX m5)

Both point and mark are set to the cursor. This command is normally bound to the pressing of button 2.

19.305. xt-mouse-point (ˆX m[01249])

Point is set to the cursor. This command is normally bound to the down-click of button 1.

19.306. xt-mouse-up (ˆX m6)

As the name implies, this command is normally bound to the release of any button (XTerm does not specify which button was released). Note that a normally configured XTerm will not pass on mouse events if the CTRL or SHIFT keys are pressed. Point is set to the cursor. If the most recently pressed button was button 1 and the CTRL key was down (and not the SHIFT key), this command performs a *yank*. If the most recently pressed button was button 2 and the CTRL key was down, this command performs a *copy-region*. If the most recently pressed button was button 2 and the CTRL and SHIFT keys were down, this command performs a *kill-region*.

19.307. yank (ˆY)

This inserts the text at the front of the kill ring (as set by an earlier *copy-region*, *kill-region*, etc.) at point. When you do multiple kill commands in a row, they are merged so that the *yank* command yanks back all of them.

19.308. yank-pop (ESC Y)

JOVE has a kill ring on which the last sixteen kills are stored. This command yanks back previous texts from the kill ring. *yank* yanks a copy of the text at the front of the ring. If you want one of the last sixteen kills you then use *yank-pop* which rotates the ring so another different entry is now at the front. You can use *yank-pop* only immediately following a *yank* or another *yank-pop*. If you supply a negative numeric argument the ring is rotated the other way. If you use this command enough times in a row you will eventually get back to where you started.

Table of Contents