CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

Hemlock User's Manual

Rob MacLachlan

22 Aug 84

Abstract

This document describes the Hemlock text editor, as of version 0.99(24). Hemlock is a customizable, extensible text editor whose initial command set closely resembles that of ITS/TOPS-20 EMACS. Hemlock is written in the SPICE LISP implementation of COMMON LISP, and can be ported to other implementations.

Spice Document S178

Keywords and index categories: <not specified>

Location of machine-readable file: >Sys>user>Ram>user.mss;1

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Projects Agency or the U.S. Government.

Table of Contents

Chapter One
Introduction

Hemlock is a text editor which follows in the tradition of EMACS and the Lisp Machine editor ZWEI. In its basic form, Hemlock has almost the same command set as ITS/TOPS-20 EMACS[1], and similar features such as multiple windows and extended commands, as well as built in documentation features. The reader should bear in mind that whenever some powerful feature of Hemlock is described, it has probably been directly inspired by EMACS.

This manual describes Hemlock's commands and other user visible features, and then goes on to tell how to make simple customizations. For complete documentation of the Hemlock primitives with which commands are written, the *Hemlock Command Implementor's Manual* is also available.

1.1 The Point and The Cursor

The *point* is the focus of editing activity at any given point in time. When text is typed in, it is inserted at the point. Nearly all commands use the point as a indication of what text to examine or modify. Textual positions in Hemlock are between characters. This may seems a bit curious at first, but it is really a necessity, since text must be inserted between characters. Although the point points between characters, it is sometimes said to point *at* a character, in which case the character after the point is referred to.

The *cursor* is the rectangular blotch that is displayed on the screen. The cursor is usually displayed on the character which is immediately after the point, but it may be displayed in other places, and is not displayed at all when computation is in progress. Wherever the cursor is displayed it indicates the current focus of attention. When input is being prompted for in the echo area, the cursor is displayed where the input is to go.

1.2 Notation

There are a number of notational conventions used in this manual which need some explanation.

---

1 In this document, "Emacs" refers to this, the original version, rather than to any of the large numbers of text editors inspired by it which may go by the same name.

1.2.1 Characters

Characters that are typed on the keyboard are printed in a **Bold Face** font. Characters such as A and # are typed in the normal fashion; others need more explanation. Characters in Hemlock have *bits*, flags that can indicate a special interpretation for that character. Although internally Hemlock can handle arbitrary combinations of up to four bits, limitations of the Perq keyboard prevent more than two bits from being used, and then only one at a time.

control — A character *char* with the control bit on is represented as C-*char*. Except for a number of arbitrary restrictions imposed by the hardware, any character can be typed as a control character by holding down the control key while typing the character.

meta — A character *char* with the meta bit on is represented as M-*char*. Since the Perq has no meta key, Hemlock simulates it by turning uppercase control characters into meta characters. To type M-A, hold down both the control and shift keys while typing A. It is therefore not possible to type M-^, for example, since this is interpreted as C-^. On the Perq2 keyboard, the characters transmitted by the keypad are turned into meta characters.

Some characters such as **Help**, **Return** and **Oops** are not printable, and thus are represented by their name, which usually corresponds to the legend on the keyboard. The down and up transitions of the left, middle and right mouse buttons are named **Leftdown**, **Leftup**, **Middledown** and so on.

1.2.2 Commands

Nearly everything that can be done in Hemlock is done using a command. Since there are many things worth doing, Hemlock provides many commands, currently nearly two hundred. Most of this manual is a description of what commands exist, how they are invoked, and what they do. This is the format of a command's documentation:

Sample Command *Command* C-Z M-Q, C-'

This command's name is Sample Command, *and it is bound to* C-Z M-Q *and* C-', *meaning that typing either of these will invoke it. After this header comes a description of what the command does:*

This command replaces all occurrences following the point of the string "`Pascal`" with the string "`Lisp`". If a prefix argument is supplied, then it is interpreted as the maximum number of occurrences to replace. If the prefix argument is negative then the replacements are done backwards from the point.

Toward the end of the description there may be information primarily of interest to

customizers and command implementors. If you don't understand this information, don't worry, the writer probably forgot to speak English.

Arguments:

target	The string to replace with `"Lisp"`.
buffer	The buffer to do the replacement in. If this is `:all` then the replacement is done in all buffers.

1.2.3 Hemlock Variables

Hemlock variables supply a simple customization mechanism by permitting commands to be parameterized. For details see 9.3.

Sample Variable *Hemlock Variable* 36

The name of this variable is Sample Variable *and its initial value is 36.*

This variable sets a lower limit on the number of replacements that be done by Sample Command. If the prefix argument is supplied, and smaller in absolute value than Sample Variable, then the user is prompted as to whether that small a number of occurrences should be replaced, so as to avoid a possibly disastrous error.

1.3 Invoking Commands

In order to get a command to do its thing, it must be invoked. The user can do this two ways, by typing the *key* to which the command is *bound* or by using an *extended command.* Commonly used commands are invoked via their key bindings since they are faster to type, while less used commands are invoked as extended commands, since they are easier to remember.

1.3.1 Key Bindings

A key is a short, usually one or two character, sequence typed on the keyboard. When a command is bound to a key, typing the key causes the command to be immediately invoked. When the command finishes doing whatever it wants to do, another key is read, and the process repeated.

Some commands read characters from the keyboard and interpret them however they please. When this

is done, key bindings have no effect, but you can invariably get out of such a state by typing **C-G** (see 1.11), and can usually find out what options are available by typing **Help** (see 1.5).

The user can easily make key bindings, altering old bindings or binding commands not previously bound. See 9.2.

In addition to the key bindings explicitly with each command, there are a number implicit key bindings created by using key links[2]. These bindings are not displayed by documentation commands such as Where Is. Here are the rules which determine what implicit key bindings exist:

- Case is not significant in key bindings, thus for each uppercase character there is an implicit binding to the corresponding lowercase character. Note however, that control-shift-*letter* produces M-*letter*, and thus is not equivalent to control-*letter*.
- A binding to M-*character* implies a binding to Alt *character*. This is for compatibility with versions of EMACS which run on conventional terminals.
- A binding to C-Z *character* implies a binding to C-Z C-*character*. In EMACS, C-Z means set both the control and meta bits, thus these two things are treated as the same, the second version setting the control bit twice. The latter version is also often easier to type.

1.3.2 Extended Commands

A command is invoked as an extended command by typing its name to the Extended Command command, which is invoked using its key binding, **M-X**.

Extended Command *Command* **M-X**

This command prompts in the echo area for the name of a command, and then invokes that command. The prefix argument is passed through to the command invoked. The command name need not be typed out in full, as long as enough of its name is supplied to uniquely identify it. Completion is available using Alt and **Space**, and a list of possible completions is given by **Help**.

1.4 The Echo Area

The echo area is the is the region which occupies the bottom few lines on the screen. It is used for two purposes: displaying brief messages to the user, and prompting.

---

2 Key links are documented in the *Hemlock Command Implementor's Manual.*

When a command needs some information form the user, it requests it by displaying a *prompt* in the echo area. The general format of a prompt is a one or two word description of the input requested, possibly followed by a *default* in brackets. Here is a typical prompt:

```
Select Buffer: [Teco Mid /Sys/Emacs/]
```

There are three general kinds of prompts:

character The response is a single character, and no confirming **Return** is needed.

keyword The response is a selection from one of a limited number of choices. Completion is available using **Space** and **Alt**, and only enough of the keyword need be typed to distinguish it from any other choice. In some cases the input need not be one of the known keywords, indicating that a new entry should be created. If this is the case then the keyword must be entered in full or completed using **Alt** so as to distinguish entering an old keyword from making a new one which is a prefix of an old one.

string The response is a string which must satisfy some property, such as being the name of an existent file.

These characters have special meanings when prompting:

Return Confirm the current parse. If no input has been entered then use the default. If for some reason the input is unacceptable, the screen is flashed, and the user given a chance to fix the problem.

Help Print some sort of help message. If the parse is a keyword parse then print all the possible completions of the current input in a pop-up window.

Alt Attempt to complete the input to a keyword parse as far as possible, flashing the screen if the result is ambiguous.

Space In a keyword parse, attempt to complete the input up to the next space. This is useful for completing the names of Hemlock commands and similar things without flashing the screen a lot, for example, Forward Word can be invoked as an extended command by typing **M-X f o Space w Return**.

C-I In a string or keyword parse, insert the default so that it may be edited.

C-P Retrieve from a history of echo area inputs, the text of the last string input. Repeating this moves to successively earlier inputs.

C-N Go the other way in the echo area history.

C-Q Quote the next character so that it is not interpreted as a command.

1.5 Online Help

Hemlock has a fairly good online documentation facility. Brief documentation for every command, variable, character attribute, and many other things, can be obtained simply by typing a command.

Help *Command* **Help, C-_**

This command dispatches to a number of other documentation commands, on the basis of a single-character command:

A	List commands and other things whose names contain a specified keyword.
D	Give the documentation for a specified command.
G	Give the documentation for any Hemlock thing.
C	Describe the command bound to some key.
L	List the last sixty characters typed.
W	List all the key bindings for a specified command.
T	Describe a LISP object.
Q	Quit without doing anything.
Help	List all of the options and what they do.

What Lossage *Command* **Help l**

This command displays the last sixty characters typed. This can be useful, if, for example, you are curious what the command was that you typed by accident.

Where Is *Command* **Help w**

This command prompts for the name of a command, and lists all the key bindings for it, and what environment they are available in, in a pop-up window.

Apropos *Command* **Help a**

This command lists all of the commands, variables and character attributes whose name contain a specified string. Brief documentation is printed for each thing, and the bindings of commands and values of variables are printed as well.

Describe Command *Command* **Help d**

This command prompts for a command, and prints its full documentation.

Describe Key *Command* **Help c**

This command prints full documentation for the command which is bound to the specified key in the current environment.

Generic Describe *Command* **Help g**

This command prints full documentation for any thing that has documentation. It first prompts for the kind of thing to document, the following options being available:

attribute	Describe a character attribute, given its name.
command	Describe a command, given its name.
key	Describe a command, given a key to which it is bound.
variable	Describe a variable, given its name. This is the default.

1.6 Display Conventions

There are two ways that Hemlock displays information on the screen, one is normal *buffer display*, in which the text being edited is shown on the screen, and the other is a *pop-up window.*

1.6.1 Pop-Up Windows

Some commands print out information that is of little permanent value. Such commands use a pop-up window to display the information. It is known as a pop-up window, because it "pops up" on the screen, overlaying text that may already be on the screen, and then goes away once the text has been read.

When the output is complete, the command displays the string `"--Flush--"` at the bottom of the output, indicating that the text may be flushed by typing Space. If any character other than space is typed, then the pop-up window will still go away, but the character will be re-read as well, and thus be interpreted as a command.

If the amount of output is too great to fit in the size of pop-up window that was created, then the message `"--More--"` will be displayed after each window full. Typing Space will go on to the next window full, while C-G aborts the remaining output.

1.6.2 Buffer Display

If a line is too long to fit within the screen width it is *wrapped*, consecutive pieces of the line being displayed on as many lines of the screen as needed to hold it. The fact that a line is wrapped is indicated by the presence of the line wrap character, currently "!", in the last column of each wrapped line. It is possible for a line to wrap off the bottom of the screen or on to the top. Hemlock wraps on the last character on the line instead of the second-to-last, as almost everyone else does. This means, among other things, that there is always at least two characters on the extension of a wrapped line. When the cursor is at the end of a line which is the full width of the screen, it is displayed at the last column, since it obviously cannot be displayed off the edge.

Most characters are displayed as themselves, but some are treated specially:

- Tabs are treated as tabs, with eight character tab-stops.
- ASCII control characters are printed as ^*char*, thus a formfeed is ^L.
- Characters with the most-significant bit on are printed as <*hex-code*>, e.g. <E2>.

Since a character may be displayed using more than one printing character, there are some positions on the screen which are in the middle of a character. When the cursor is on a character with a multiple-character representation, it will always be displayed on the first character.

1.7 The Prefix Argument

The prefix argument is an integer argument which may be supplied to a command. It is known as the prefix argument because it is specified by invoking some prefix argument setting command immediately before the command to be given the argument. The following statements about the interpretation of the prefix argument are true:

- When it is meaningful, most commands interpret the prefix argument as a repeat count, causing the same effect as invoking the command that many times.
- When it is meaningful, most commands that use the prefix argument interpret a negative prefix argument as meaning the same thing as a positive argument, but the action is done in the opposite direction.
- Most commands treat the absence of a prefix argument as meaning the same thing as a prefix argument of one.
- Many commands ignore the prefix argument entirely.

- Some commands do none of the above.

The following commands are used to set the prefix argument:

Argument Digit *Command* C-*digit*, M-*digit*

Typing a number using this command sets the prefix argument to that number, for example, typing C-1 C-2 sets the prefix argument to twelve.

Negative Argument *Command* C--

This command negates the prefix argument, or if there is none, sets it to negative one. For example, typing C-- C-7 sets the prefix argument to negative seven.

Universal Argument *Command* C-U

This command sets the prefix argument or multiplies it by four. If digits are typed immediately afterward, they are echoed in the echo area, and the prefix argument is set to the specified number. If no digits are typed then the prefix argument is multiplied by four. C-U - 7 sets the prefix argument to negative seven. C-U C-U sets the prefix argument to sixteen. C-4 C-2 C-U sets the prefix argument to one hundred and sixty-eight. C-U C-0 sets the prefix argument to forty.

Universal Argument Default *Hemlock Variable* 4

This variable determines the default value and multiplier for the Universal Argument command.

1.8 Modes

A mode provides a way to change Hemlock's behavior by specifying a modification to current key bindings, values of variables, and other things. Modes are typically used to adjust Hemlock to suit a particular editing task, e.g. Lisp mode is used for editing LISP code.

Modes in Hemlock are not like modes in most text editors, such as insert or alter mode in SOS, and in fact Hemlock really a "modeless" editor. There are two ways that the Hemlock mode concept differs from the conventional one:

1. Modes do not usually alter the environment in a very big way, i.e. replace the set of commands bound with another totally disjoint one. When a mode redefines what a key does, it usually redefined to have a slightly different meaning, rather than a totally different one. For this reason, typing a given key does pretty much the same thing no matter what modes are in effect. This property is the distinguishing characteristic of a modeless editor.

2. Once the modes appropriate for editing a given file have been chosen, they are seldom, if ever, changed. One of the advantages of modeless editors is that time is not wasted changing modes.

A *major mode* is used to make some big change in the editing environment. Language modes such as Pascal mode are major modes. A major mode *name* is usually turned on by invoking the command *mode-name* Mode as an extended command. There is only one major mode present at a time. Turning on a major mode turns off the one that is currently in effect.

A *minor mode* is used to make a small change in the environment, such as automatically breaking lines if they get too long. Unlike for major modes, any number of minor modes may be present at once. Ideally minor modes should do the "right thing" no matter what major and minor modes are in effect, but this is not currently the case when key bindings conflict. If, for example, the major mode is Lisp and Fill is a minor mode, then typing **Linefeed** will fill the line, but not do LISP indentation.

Modes can be envisioned as switches, the major mode corresponding one big switch which is thrown into the correct position for the type of editing being done, and each minor mode corresponding to an on-off switch which controls whether a certain characteristic is present.

1.9 The Modeline

The modeline is the line displayed in inverse video at the bottom of each window. This line is used to display information about the buffer displayed in that window. Here is a typical modeline:

```
Hemlock  (Fundamental Fill)  /sys/slisp/hemlock/user.mss#1
```

This tells us that the file associated with this buffer is `"/sys/slisp/hemlock/user.mss#1"` and the modes currently present are Fundamental and Fill. The major mode is always displayed first, followed by any minor modes. If the buffer has no associated file, then the buffer name will be displayed instead. If such a buffer's name was Silly, then it would be displayed in the following fashion:

```
Hemlock  (Lisp)  Silly:
```

1.10 Entering and Exiting

Hemlock is entered by using the LISP `ed` function. Simply typing `(ed)` will enter Hemlock leaving you in the state that you were in when you left it. If Hemlock has never been entered before then the current buffer will be Main.

`ed` may be given an argument which may either be a file name or a symbol. Typing (`ed` *filename*) will cause the specified file to be read into Hemlock, as though by Find File. Typing (`ed` *symbol*) will pretty-print the definition of the symbol into a buffer whose name is obtained by concatenating "`Edit `" to the beginning of the symbol's name.

Exit Hemlock *Command* C-C, C-X C-Z

This command exits Hemlock, returning `T`. Exit Hemlock does not by default save modified buffers, or do anything else that you might think it should do, it simply exits. After exiting, you may reenter at any time without having lost anything by typing (`ed`) to LISP. Before you quit from LISP using (`quit`), you should save any modified files that you want to be saved.

1.11 Helpful Information

This section contains assorted helpful information which may be useful in staying out of trouble, or lacking that, getting out of trouble.

- It is possible to get some sort of help nearly everywhere by typing **Help**.
- Various commands take over the keyboard and insist that you type the things that they want to hear. If you get in such a situation and want to get out, you can invariably do so by typing **C-G** some small number of times. If this fails you can try typing C-C to exit Hemlock and then "(`ed`)" to reenter it.
- It is a good idea to get into the habit of saving your changes periodically so that you will not lose much work if the system crashes.
- If you save a buffer whenever you leave it, it is less likely that you will forget to write out changed buffers.
- Before you quit, *always* do a **C-X C-B** to see if there are any buffers which need to be written out.
- If the screen changes unexpectedly, you may have accidentally typed an incorrect command. Use **Help l** to see what it was. If you are not familiar with the command, use **Help c** to see what is so that you know what damage has been done. Many interesting commands can be found in this fashion. This is an example of the much-underrated learning technique known as "Learning by serendipitous malcoordination". Who would ever think of looking for a command that deletes all files in the current directory?
- If you accidentally type a "killing" command such as **C-W**, you can get the lost text back using **C-Y**.

1.12 Recursive Edits

Some sophisticated commands, such as Query Replace, will, at your request, place you in a *recursive edit*. A recursive edit is simply a recursive invocation of Hemlock done within some command. A recursive edit is useful because it enables arbitrary editing to be done during the execution of a command without losing any state that that the command might have. Once the recursive edit is exited, the command that did it proceeds as though nothing had happened. Hemlock indicates that you are in a recursive edit by putting a "[" before and a "]" after the modeline in the current window. Nested recursive edits will cause nested square-brackets to be displayed around the modeline.

Exit Recursive Edit *Command* C-Z z

This command exits the current recursive edit, returning `nil`. If invoked when not in a recursive edit, then Hemlock will exit, `ed` returning `nil`.

Abort Recursive Edit *Command* C-]

This command causes the command which invoked the recursive edit to get an error. If Abort Recursive Edit is invoked when not in a recursive edit, then Hemlock will exit, `ed` returning the string `"Recursive edit aborted."`.

1.13 User Errors

If you use a command and the screen flashes, possibly with a message such as `"No next line."` appearing in the echo area, then you have attempted to do something that Hemlock does not want to do. You had best try something else, since Hemlock, being far more stupid than you, is far more stubborn. Since Hemlock is an extensible editor, another alternative is to change the command that complained to do what you wanted it to do.

1.14 Internal Errors

A message of this form may appear in the echo area, accompanied by a flash of the screen:

```
Internal error:
Wrong type argument, NIL, should have been of type SIMPLE-VECTOR.
```

If the error message is a file related error such as the following, then you have probably done something illegal which Hemlock did not catch, but was detected by the file system.

```
Internal error:
No access to "/lisp2/emacs/teco.mid"
```

Otherwise, you have found a bug. Try to avoid the behavior that resulted in the error, and report the problem to you system maintainer. Since LISP has fairly robust error recovery mechanisms, probably no damage has been done.

If a truly abominable error from which Hemlock cannot recover occurs, then you will be thrown into the LISP debugger. At this point it would be a good idea to use `save-all-buffers` to save any changes, and then start a new LISP.

The LISP function `save-all-buffers` may be used to recover from a seriously broken Hemlock. To use this, simply type "`(save-all-buffers)`" to the top-level ("`* `") or break-loop ("`1> `") prompt and answer the questions is asks. *Do not call this function while in Hemlock.*

Chapter Two
Basic Commands

2.1 Motion Commands

There is a fairly small number of basic commands for moving around in the buffer. While there are many other more complex motion commands, these are by far the most commonly used and the easiest to learn.

Forward Character *Command* **C-F, Rightarrow**
Backward Character *Command* **C-B, Leftarrow**

Forward Character moves the point forward by one character. If a prefix argument is supplied, then the point is moved by that many characters. Backward Character is identical, except that it moves the point backwards.

Forward Word *Command* **M-F**
Backward Word *Command* **M-B**

These commands move the point forward and backward over words. The point is always left between the last word and first non-word character in the direction of motion. This means the after moving backward the cursor appears on the first character of the word, while after moving forward, the cursor appears on the delimiting character. Supplying a prefix argument moves the point by that many words.

Next Line *Command* **C-N, Downarrow**
Previous Line *Command* **C-P, Uparrow**

These commands are used to move to adjacent lines, while remaining the same distance within a line. Note that this motion is by logical lines, each of which may take up many lines on the screen if it wraps. If a prefix argument is supplied, then the point is moved by that many lines.

The position within the line at the start is recorded, and each successive use of **C-P** or **C-N** attempts to move the point to that position on the new line. If it is not possible to move to the recorded position because the line is shorter, then the point is left at the end of the line.

End of Line *Command* C-E
Beginning of Line *Command* C-A

End of Line moves the point to the end of the current line, while Beginning of Line moves to the beginning. If a prefix argument is supplied, then the point is moved to the end or beginning of the line that many lines below the current one.

Scroll Window Down *Command* C-V
Scroll Window Up *Command* M-V

Scroll Window Down moves forward in the buffer by one screenfull of text, the exact amount being determined by the size of the window. If a prefix argument is supplied, then the screen is scrolled by that many lines. When the point is moved off of the screen, it is moved to the vertical center of the new screen. Scroll Window Up is identical to Next Screen, except that it moves backwards.

Screen Overlap *Hemlock Variable* 2

This variable is used by Next Screen and Previous Screen to determine the number of lines by which the new and old screen should overlap.

End of Buffer *Command* C->, Alt >
Beginning of Buffer *Command* C-<, Alt <

These commands are used to conveniently get to the very beginning and end of the text in a buffer. Before the point is moved, its position is saved by pushing it on the position of the mark stack (see 2.3).

2.1.1 Using The Mouse

Especially when moving large distances, it can be convenient to use the mouse to point to positions in text. Hemlock defines several commands for using the mouse.

Here to Top of Window *Command* Rightdown
Top Line to Here *Command* Leftdown

Here to Top of Window scrolls the window so as to move the line which is under the mouse cursor to the top of the window. This has the effect of moving forward in the buffer by the distance of the mouse cursor from the top of the window. Top Line to Here is the inverse operation, it scrolls backward, moving current top line underneath the mouse.

If the mouse is pointing into a window other than the current one, then that window is switched to before the scrolling is done.

Point to Here *Command* **Middledown**

This command moves the point to the position of the mouse, changing to a different window if necessary.

2.2 Other Motion Commands

Back to Indentation *Command* **M-M**

Move point to the first non-whitespace character on the current line.

2.3 The Mark and The Region

Each buffer has a distinguished position known as the *mark*, which starts at the beginning of the buffer. The area between the mark and the point is known as the *region*. Many Hemlock commands which manipulate large pieces of text use the text in the region. Neither the region nor the mark are visible, so the only way to be sure that the mark is in a particular place is to move it there. This is usually not a problem, the mark is usually in the wrong place anyway, so normal practice is to set it immediately before using it.

Due to the way the region is defined, it always exists, even if the user has not set any marks. Accidentally typing a command which uses the region when it has not been meaningfully defined is a common source of mysterious, catastrophic damage to your text.

Push Mark/Point to Here *Command* **Middleup**

This is a mouse command which is used to set the mark, and thus define the region. Note that this command is bound to **Middleup**, that is releasing the middle button, while Point to Here is bound to **Middledown**. To use these two commands to mark out a region, press down the middle button at one end of the region and release it at the other. The mark is left at the place the button is pressed, and the point at the place it is released.

What this command actually does is move the point to the position of the mouse, pushing the old position of the point on the mark stack if it was different.

Exchange Point and Mark *Command* **C-X C-X**

Exchange Point and Mark interchanges the position of the point and the mark, thus moving to where the mark was, and leaving the mark where the point was. This command can be used to switch between two positions in a buffer, since repeating it undoes its effect. Unlike other mark-modifying commands, this does not push the old mark on the mark stack.

Mark Whole Buffer *Command* C-X h

This command sets the region around the whole buffer. If a prefix argument is supplied, then the mark is put at the beginning and the point at the end. The mark is pushed on the mark stack beforehand, so popping twice will restore it.

2.3.1 The Mark Stack

As was hinted at earlier, there is actually a stack of marks. The current mark is determined by the mark which is on the top of the stack, but it is possible to recover earlier values of the mark by popping marks off of this stack.

Set/Pop Mark *Command* C-@

With no prefix argument, C-@ sets the mark to the current location of the point. The use of the prefix argument by this command is slightly bizzare. If the prefix argument is four the mark is popped into the point, meaning that the point is moved to the mark, and the mark is moved to the value before it on the mark stack. If the prefix argument is sixteen, then the mark stack is popped without affecting the point.

The reason for these particular values for the prefix argument is that they can be easily generated using the Universal Argument command by typing C-U or C-U C-U. This command examines the variable Universal Argument Default so that this will still work even if the default value is changed.

2.4 Modification Commands

There is a wide variety of basic text-modification commands, but once again the simplest ones are the most often used.

2.4.1 Inserting Characters

In Hemlock, printing characters may be inserted by simply typing them, while others require extra effort. Like everything else in Hemlock, basic text insertion is implemented by commands.

Self Insert *Command*

Self Insert inserts the character which was typed to invoke it into the buffer. This command is normally bound all printing characters and Space. If a prefix argument is supplied, then the character is inserted that many times.

New Line *Command* **Return**

This command, which has roughly the same effect as inserting a **Newline**, is used to move onto a new blank line. If there are at least two blank lines beneath the current one then **Return** cleans off any whitespace on the next line and uses it, instead of inserting a newline. This behavior is desirable when inserting in the middle of text, because the bottom half of the screen does not scroll down each time **New Line** is used.

Quoted Insert *Command* **C-Q**

Many characters cannot be inserted by **Self Insert** because they are bound to another command, or are otherwise magical (**C-G, Help**). **C-Q** gets around this problem by reading a character from the keyboard with any special interpretation inhibited. A common use for this command is to insert a **Formfeed** by doing **C-Q C-L**. If a prefix argument is supplied, then the character is inserted that many times.

Open Line *Command* **C-O**

This command inserts a newline into the buffer without moving the point. This command may also be given a prefix argument to insert a number of newlines, thus opening up some room to work in the middle of a screen of text.

2.4.2 Deleting Characters

There are a number of commands for deleting characters as well. One should avoid giving numeric arguments to these commands, since once the text is deleted it is gone forever.

Delete Next Character *Command* **C-D**
Delete Previous Character *Command* **Delete, Backspace, C-H**

Delete Next Character deletes the character immediately following the point, that is, the character which appears under the cursor. When given a prefix argument, **C-D** deletes that many characters after the point. **Delete Previous Character** is identical, except that it deletes characters before the point.

Delete Previous Character Expanding Tabs *Command*

Delete Previous Character Expanding Tabs is identical to **Delete Previous Character**, except that it treats tabs as the equivalent number of spaces. Various language modes that use tabs for indentation bind **Delete** to this command.

2.4.3 Killing and Deleting

Hemlock has many commands which kill text. Killing is a variety of deletion which saves the deleted text for later retrieval. The killed text is saved in a ring buffer known as the *kill ring*. Killing has two main advantages over deletion:

1. If text is accidentally killed, a not uncommon occurrence, then it can be restored.
2. Text can be moved from one place to another by killing it and then restoring it in the new location.

Killing is not the same as deleting. When a command is said to delete text, the text is permanently gone, and is not pushed on the kill ring. Commands which delete text generally only delete things of little importance, such as single characters or whitespace.

2.4.4 Kill Ring Manipulation

Un-Kill *Command* **C-Y**

This command "yanks" back the most recently killed piece of text, leaving the mark before the inserted text and the point after. If a prefix argument is supplied then the text that distance back in the kill ring is yanked.

Rotate Kill Ring *Command* **M-Y**

This command rotates the kill ring forward, replacing the most recently yanked text with the next most recent text in the kill ring. **M-Y** may only be used immediately after a use of **C-Y** or a previous use of **M-Y**. This command is used to step back through the text in the kill ring if the desired text was not the most recently killed, and thus could not be retrieved directly with a **C-Y**. If a prefix argument is supplied, then the kill ring is rotated that many times.

Kill Region *Command* **C-W**

This command kills the text between the point and mark, pushing it onto the kill ring. This command is usually the best way to move or remove large quantities of text.

Save Region *Command* **M-W**

This command pushes the text in the region on the kill ring, but doesn't actually kill it, giving an effect similar to typing **C-W C-Y**. This command is useful for duplicating large pieces of text.

2.4.5 Killing Commands

Most commands which kill text append into the kill ring, meaning that consecutive uses of killing commands will insert all text killed into the top entry in the kill ring. This allows large pieces of text to be killed by repeatedly using a killing command.

Kill Line *Command* **C-K**

This command kills the text from the point to the end of the current line, deleting the line if it is empty. If a prefix argument is supplied, then that many lines are killed. Note that prefix argument is not the same as a repeat count.

Kill to Beginning of Line *Command* **Oops**

Oops kills the text from the point to the beginning of the current line.

Kill Next Word *Command* **M-D**
Kill Previous Word *Command* **C-Backspace, Alt Backspace, Alt Delete**

Kill Next Word kills from the point to the end of the current or next word. If a prefix argument is supplied, then that many words are killed. Kill Previous Word is identical, except that it kills backward.

2.4.6 Case Modification Commands

Hemlock provides a few case modification commands, which are often useful for correcting typos.

Capitalize Word *Command* **M-C**
Lowercase Word *Command* **M-L**
Uppercase Word *Command* **M-U**

These commands modify the case of the characters from the point to the end of the current or next word, leaving the point after the end of the word affected. A positive prefix argument modifies that many words, moving forward. A negative prefix argument modifies that many words before the point, but leaves the point unmoved.

2.4.7 Transposition Commands

Hemlock provides a number of transposition commands. A transposition command swaps the "things" before and after the point and moves forward one "thing". Just how a "thing" is defined depends on the particular transposition command. Transposition commands, particularly Transpose Characters and

transpose Words, are useful for correcting typos. More obscure transposition commands can be used to amaze you friends and demonstrate your immense knowledge of exotic EMACS commands.

To the uninitiated, the behavior of transposition commands may seem mysterious, and some implementors have attempted to "improve" the definition of transposition, but the true EMACS definition used in Hemlock has two useful properties:

1. Repeated applications of a transposition command have a useful effect. The way to visualize this effect is that each use of the transposition command drags the previous thing over the next thing. It is possible to correct double transpositions easily using Transpose Characters.
2. Transposition commands move backward with a negative prefix argument, thus undoing the effect of the equivalent positive argument.

Transpose Characters *Command* C-T
This command exchanges the characters on either side of the point and moves forward, unless at the end of a line, in which case it transposes the previous two characters without moving.

Transpose Lines *Command* C-X C-T
This command transposes the previous and current line, moving down to the next line. With a zero argument, it transposes the current line and the line the mark is on.

Transpose Words *Command* M-T
This command transposes the previous word and the current or next word With a zero argument, it transposes the words at the point and mark.

2.4.8 Whitespace Manipulation

Just One Space *Command* C-|
This command deletes all blank characters before and after the point, and then inserts one space. If a prefix argument is supplied, then that number of spaces is inserted.

Delete Horizontal Space *Command* C-\, Alt \
This command deletes all blank characters around the point.

Delete Indentation *Command* C-^, Alt ^
Delete Indentation joins the current line with the previous one, deleting excess whitespace. This operation is the inverse of the Linefeed command in most modes.

One space is left between the two joined line fragments, unless the Lisp Syntax attribute of first non-blank character on the second line is `:close-paren`.

Indent Rigidly *Command* C-X Tab, C-X C-I

This command changes the indentation of all the lines in the region. Each line is moved to the right by the number of spaces specified by the prefix argument, which defaults to eight. A negative prefix argument moves lines left.

2.5 Filtering

Filtering is a simple way to perform a fairly arbitrary transformation on text. Filtering text replaces the string in each line with the result of applying a LISP function of one argument to that string. The function must neither destructively modify the argument nor the return value. It is an error for the function to return a string containing newline characters.

Filter Region *Command*

This function prompts for an expression which is evaluated to obtain a function to be used to filter the text in the region. For example, to capitalize all the words in the region one could respond:

```
Function: #'string-capitalize
```

Since the function may be called many times, it should probably be compiled. Functions for one-time use can be compiled using the compile function as in the following example which removes all the semicolons on any line which contains the string "PASCAL":

```
Function: (compile nil '(lambda (s)
                          (if (search "PASCAL" s :test #'char-equal)
                              (remove #\; s)
                              s)))
```

2.6 Searching and Replacing

Searching for some string known to appear in the text is a commonly used method of moving long distances in a file. Replacing occurrences of one pattern with another is a useful way to make many simple changes to text. Hemlock provides powerful commands for doing both of these operations.

Default Search Kind *Hemlock Variable* `:string-insensitive`

This variable determines the kind of search done by searching and replacing commands. There are currently two useful values for this variable:

`:string-insensitive`	Do a case-insensitive string search.
`:string-sensitive`	Do a case-sensitive string search.

Incremental Search *Command* **C-S**
Reverse Incremental Search *Command* **C-R**

Incremental Search searches an occurrence of a string somewhere after the current location of the point. It is known as an incremental search because it reads characters form the keyboard one at a time, and immediately searches for the pattern it has read so far. This is useful because it is possible to initially type in a very short pattern, and then add more characters if it turns out that this pattern has too many spurious matches.

The following characters are interpreted as commands:

C-S	Search forward for an occurrence of the current pattern. This can be used repeatedly to skip from one occurrence of the pattern to the next, or can be used to change the direction of the search if it is currently a reverse search. If C-S is typed when the search string is empty, then a search is done for the string that was used by the last searching command.
C-R	Similar to C-S, except that it searches backwards.
Delete, Backspace	Undoes the effect of the last character typed. If that character simply added to the search pattern, then it removes the character from the pattern, moving back to the first match for that string. If the character was a C-S or C-R then the previous match is skipped back to, and the search direction possibly reversed.
C-G	If the search is currently failing, meaning that there is no occurrence of the search pattern in the direction of search, the C-G deletes enough characters off of the end of the pattern to make it successful. If the search is currently successful, then C-G causes the search to be aborted, leaving the point where it was when the search started.
Alt	Exit at the current position in the text, unless the search string is empty, in which case a non-incremental string search is entered.
C-Q	Search for the next character, rather than treating it as a command.

If any non-printing, unquoted character other than those described above is typed, then the

search is exited and the character is processed again with its normal interpretation. Typing **C-A** will exit the search *and* go to the beginning of the line.

Incremental Search Exit Char *Hemlock Variable* t

If this variable is false, then the action of **Alt** in Incremental Search is inhibited. Typing **Alt** will then still cause the command to exit, but the character will be read again as a command.

Forward Search *Command*
Reverse Search *Command*

These commands do a normal dumb string search, prompting for the search string in a normal dumb fashion. One reason for using a non-incremental search is that it may be faster, since it is possible to specify a long search string from the very start. Since Hemlock uses the Boyer–Moore search algorithm, the speed of the search increases with the size of the search string.

Query Replace *Command* **C-%, Alt %**

This command prompts in the echo area for a target string and a replacement string, then searches for an occurrence of the target following the point. When a match is found, any of a number of actions may be taken, depending on a single character command read from the keyboard. The following characters are used by Query Replace:

Space, Y	Replace this occurrence of the target with the replacement string, and search again.
Delete, Backspace, N	Do not replace this occurrence, but continue the search.
!	Replace this and all remaining occurrences without prompting again.
.	Replace this occurrence and exit.
C-R	Go into a recursive edit (see 1.12) at the current location. The search will be continued from wherever the point is left when the recursive is exited. This is useful for handling more complicated cases where a simple replacement will not achieve the desired effect.
Alt	Exit without doing any replacement.
Help	Print a list of all the options available.

Any other character causes the command to exit, unreading the character, and thus causing it to be reinterpreted as a normal command.

If the replacement string is all lowercase, then a heuristic is used that attempts to make the case

of the replacement the same as that of the particular occurrence of the target pattern. If "foo" is being replaced with "bar" then "Foo" is replaced with "Bar" and "FOO" with "BAR".

Case Replace *Hemlock Variable* t

If this variable is true then the case preserving heuristic in Query Replace is enabled, otherwise all replacements are done with the replacement string exactly as specified.

Replace String *Command*

Prompts for a target and replacement string and replaces all occurrences of the target string following the point with the replacement string.

Chapter Three
Files, Buffers and Windows

3.1 Introduction

Hemlock provides three different abstractions which are used in combination to solve the text-editing problem, while other editors tend to mash these ideas together into two or even one.

File	A file provides permanent storage of text. Hemlock has commands to read files into buffers and write buffers out into files.
Buffer	A buffer provides temporary storage of text and a capability to edit it. A buffer may or may not have a file associated with it, if it does, the text in the buffer need bear no particular relation to the text in the file. In addition, text in a buffer may be displayed in any number of windows, or may not be displayed at all.
Window	A window displays some portion of a buffer on the screen. There may be any number of windows on the screen, each of which may display any position in any buffer. It is thus possible, and often useful, to have several windows displaying different places in the same buffer.

3.2 Buffers

In addition some text, a buffer has several other user-visible attributes, among them:

A name	A buffer is identified by its name, which allows it to be selected, destroyed, or otherwise manipulated.
A collection of modes	The modes present in a buffer alter the set of commands available and otherwise alter the behavior of the editor. For details see 1.8.
A modification flag	This flag is set whenever the text in a buffer is modified. It is often useful to know whether a buffer has been changed, since if it has it should probably be saved in its associated file eventually.
A write-protect flag	If this flag is true, then any attempt to modify the buffer will result in an error.

Select Buffer *Command* **C-X b**

This command prompts for the name of a buffer, and then makes that buffer the *current buffer.* The newly selected buffer is displayed in the current window, and editing commands now edit the text in that buffer. Each buffer has its own point, thus the point will be in the place it was the

last time the buffer was selected. When prompting for the buffer, the default is the buffer that was selected before the current one.

Select Previous Buffer *Command* **C-Z l**

This command selects the buffer that has most recently selected. Its effect is identical to that of **C-X b Return.**

Create Buffer *Command* **C-X M-B**

This command is very similar to **Select Buffer**, except that the buffer need not already exist. If the buffer does not exist a new, empty buffer is created with the specified name.

Kill Buffer *Command* **C-X k**

This command is used to make a buffer go away. There is no way to restore a buffer that has been accidentally deleted, so the user is given a chance to save the hapless buffer if it is modified. This command is poorly named, since it has nothing to do with killing text.

List Buffers *Command* **C-X C-B**

This command displays a list of all that buffers that exist in a pop-up window. A "*" is displayed before the name of each modified buffer, and the associated filename (or number of lines if none) is displayed after the buffer name.

Buffer Not Modified *Command* **C-~, Alt ~**

This command resets the current buffer's modification flag — *it does not save any changes.* Doing this is primarily useful in the case where a buffer was accidentally modified and the change then undone. Resetting the modified flag the indicates that the buffer has no changes that need to be written out.

Check Buffer Modified *Command* **C-X ~**

This command indicates whether the current buffer is modified.

Insert Buffer *Command*

This command prompts for the name of a buffer, the contents of which are inserted at the point. The buffer inserted is unaffected

Rename Buffer *Command*

This command prompts for a new name for the current buffer, which defaults to a name derived from the associated filename.

3.3 Files

These commands either read a file into the current buffer or write it out to some file. Various other bookkeeping operations are performed as well.

Find File *Command* **C-X C-F**

This is the command normally used to get a file into Hemlock, it prompts for the name of a file, and if that file has already been read in, selects that buffer, otherwise it reads file into a new buffer whose name is derived from the name of the file. If the file does not exist then the buffer is left empty and "`(New File)`" is displayed in the echo area; the file may then be created by saving the buffer.

The buffer name created is in the form "*name type directory*", thus the filename "`/sys/emacs/teco.mid`" would have the corresponding buffer name "`Teco Mid /Sys/Emacs/`". The reason for rearranging the fields in this fashion is that it facilitates recognition since the components most likely to differ are placed first. If the buffer cannot be created because it already exists, but has another file in it (an unlikely occurrence) then the user is prompted for the buffer to use, as by **Create Buffer**.

Find File currently does not check to see if the file has been modified since the last time it was read in, thus it is necessary to force the file to be read using **Visit File** if this is the case.

Save File *Command* **C-X C-S**

This command writes the current buffer out to its associated file and resets the buffer modification flag. If there is no associated file then the user is prompted for a file, and that is made the associated file. If the buffer is not modified, then the user is asked whether to actually write it or not.

Visit File *Command* **C-X C-V**

This command prompts for a file and reads it into the current buffer, setting the associated filename. Since the old contents of the buffer are destroyed, the user is given a chance to save the buffer if it is modified. As for **Find File**, the file need not actually exist.

Write File *Command* **C-X C-W**

This command prompts for a file and writes the current buffer out to it, changing the associated filename and resetting then modification flag.

Backup File *Command*

This command is similar to Write File, but it neither sets the associated filename nor clears the modification flag. This is useful for saving the current state somewhere else, perhaps on a reliable machine.

Insert File *Command* C-X C-R

Prompts for a file and inserts it at the point.

Add Newline at EOF on Writing File *Hemlock Variable* `:ask-user`

This variable controls whether Save File and Write File add a newline at the end of the file if the last line is non-empty.

`t`	Automatically add a newline, and tell the user it was done.
`nil`	Never add a newline.
`:ask-user`	Ask the user whether to add a newline or not.

Some programs will either lose the text on the last line, or get an error, when the last line does not have a newline at the end.

3.3.1 Filename Defaulting and Merging

When Hemlock prompts for the name of a file, a default is always offered. Unless otherwise noted, this default is the current buffer's associated filename. If there is no associated filename, then a filename is created with the current buffer's name as its name and the most recently used file type as its type.

When a default is present in prompt for a file, the input given is *merged* with the default filename. The exact semantics of merging, which is described in the COMMON LISP manual, is somewhat involved, but the general idea is that any part (device, directory, name, type or version) of the filename which is left unspecified is filled in from the defaults. This can be quite convenient, as it often eliminates the need to type in the directory and type components.

In order to get around some of the problems of merging, there are two cases which Hemlock treats specially:

1. If a file can be found using the current search list which is identical to the name entered, then no merging is done. This permits a file which is in a directory on the search list to be found when default directory is not on the search list.
2. Entering an empty file type (`"foo."`) inhibits merging in of the default type. This permits the creation of a file having no type, in this case `"foo"`.

Pathname Defaults *Hemlock Variable* `(pathname "gazonk.del")`

This variable contains a pathname which is used to supply defaults for file manipulation commands when we don't have anything better. Any command which prompts for a file should set this to the pathname of the file specified.

3.3.2 Type Hooks and File Options

When a file is read either by Find File or Visit File, Hemlock attempts to guess the correct mode to put the buffer in based on the file's *type*, the part of the filename after the last dot. Any default action may be overridden by specifying the mode in the file's *file options*.

File options are specified by a special syntax in the first line of a file. If the first line contains the string "`-*-`" then the text until the next "`-*-`", which must be on the same line, is interpreted as a list of "*attribute*: *value*" pairs separated by semicolons. A typical example:

```
;;; -*- Mode: Lisp; Package: Hemlock -*-
```

These attributes are currently defined:

Mode	The argument is the name of the mode to put the buffer in when the file is read.
Package	When in the buffer with this file in it, the LISP variable `*package*` is set to the specified package. This is only meaningful for LISP code.

If the option list contains no "`:`" then the entire string is used as the name of the mode for the buffer.

Process File Options *Command*

This command processes the file options in the current buffer as described above. This is useful when the options have been changed or when a file is created

3.4 Windows

The Hemlock window commands currently have some deficiencies, the most notable being that there is no way to change the size of a window once it is made.

New Window *Command* **C-X 2**

This command creates a new window on the screen which displays the current buffer.

Next Window *Command* C-X n
Previous Window *Command* C-X p

These commands make the window below or above the current window the new current window, wrapping around at the top and bottom of the screen. Doing so will often change the current buffer as well.

Delete Window *Command* C-X C-D, C-X d
Delete Next Window *Command* C-X 1

Delete Window makes the current window go away, growing some other window to take up the space. Delete Next Window does the same thing to the next window.

Line to Top of Window *Command* C-!, Alt !

Scroll the current window up until the current line is at the top of the screen.

Center Line *Command* C-#

Attempt to scroll the current window so as to vertically center the current line.

Scroll Next Window Down *Command* C-Z v
Scroll Next Window Up *Command* C-Z M-V

These commands are the same as Scroll Window Up and Scroll Window Down except that they operate on the next window.

Refresh Screen *Command* C-L

This command refreshes the entire screen, which is useful if got trashed somehow.

Chapter Four
Editing Documents

Although Hemlock is not dedicated to editing documents as word processing systems are, it provides a number of commands for this purpose. If Hemlock is used in conjunction with a text-formatting program, then its lack of complex formatting commands is no liability.

4.1 Sentence Commands

A sentence is defined as a sequence of characters terminated by a period, question mark or exclamation point, followed by either two spaces or a newline. A sentence may also be terminated by by the end of a paragraph. Any number of closing delimiters, such as brackets or quotes, may be between the punctuation and the whitespace. This somewhat complex definition of a sentence is used so that periods in abbreviations are not misinterpreted as sentence ends.

Forward Sentence *Command* M-A
Backward Sentence *Command* M-E

Forward Sentence moves the point forward past the next sentence end. Backward Sentence moves to be beginning of the current sentence. A prefix argument may be used as a repeat count.

Forward Kill Sentence *Command* M-K
Backward Kill Sentence *Command* C-X Delete, C-X Backspace

Forward Kill Sentence kills text from the point through to the end of the current sentence. Backward Kill Sentence kills from the point to the beginning of the current sentence. A prefix argument may be used as a repeat count.

Mark Sentence *Command*

This command puts the point at the beginning and the mark at the end of the next or current sentence.

4.2 Paragraph Commands

A paragraph may be delimited by a blank line or a line beginning with one of these characters: "@ , - ' ."; if so, that line is not part of the paragraph. A line with at least one leading whitespace character may

also introduce a paragraph, and is considered to be part of the paragraph. Any fill-prefix which is present on a line is disregarded for the purpose of locating a paragraph boundary.

Forward Paragraph *Command* Alt]
Backward Paragraph *Command* Alt [

Forward Paragraph moves to the end of the current or next paragraph. Backward Paragraph moves to the beginning of the current or previous paragraph. A prefix argument may be used as a repeat count.

Mark Paragraph *Command* M-H

This command puts the point at the beginning and the mark at the end of the current paragraph.

4.3 Filling

Filling is a coarse text-formatting process which attempts to make all the lines roughly the same length, but does not alter the amount of space between words. Editing text may leave lines with all sorts of strange lengths; filling this text will return it to a moderately aesthetic form.

Set Fill Column *Command* C-X f

This command sets the fill column to the column that the point is currently at, or the one specified by the prefix argument, if it is supplied. The fill column is the column past which no text is permitted to extend.

Set Fill Prefix *Command* C-X .

This command sets the fill prefix to the text from the beginning of the current line to the point. The fill-prefix is a string which filling commands leave at the beginning of every line filled. This feature is useful for filling indented text.

Fill Column *Hemlock Variable* 75
Fill Prefix *Hemlock Variable* nil

These variables hold the value of the fill prefix and fill column, thus setting these variables will change the way filling is done. If Fill Prefix is `nil`, then there is no fill prefix.

Fill Paragraph *Command* M-Q

This command fills the text in the current paragraph. The point is not moved.

Fill Region *Command* M-G
Fill Region Confirmation Threshold *Hemlock Variable* 50

This command fills the text in the region. Since this is good way to mangle a large quantity of text, when there are more lines in the region than the value of Fill Region Confirmation Threshold the user is asked for confirmation. This check can be inhibited by setting the variable to `nil`.

Auto Fill Mode *Command*

This command turns on or off the Fill minor mode in the current buffer. When in Fill mode, Space, Return and Linefeed are rebound to commands that check whether the point is past the fill column, and fill the current line if it is. This enables typing text without having to break the lines manually.

If a prefix argument is supplied, then instead of toggling, the sign determines whether Fill mode is turned off, a positive argument argument turns in on and a negative one turns it off.

Auto Fill Linefeed *Command* `Fill: Linefeed`
Auto Fill Return *Command* `Fill: Return`

This command fills the current line if it needs it, goes to a new line, as though by the New Line command, and the inserts the fill prefix, if any. Auto Fill Return is identical, except that it does not insert the fill prefix on the new line.

Auto Fill Space *Command* `Fill: Space`

If no prefix argument is supplied, this command inserts a space and fills the current line if it extends past the fill column. If the argument is zero, then it fills the line if needed, but does not insert a space. If the argument is positive then that many spaces are inserted without filling.

4.4 Spelling Correction

Hemlock has a spelling correction facility based on the dictionary for the ITS spell program. This dictionary is fairly small, having only thirty thousand words or so, which means it fits on your disk, but it also means that many reasonably common words are not in the dictionary. A correct spelling for a misspelled word will be found if the word is in the dictionary and is only erroneous in that it has a wrong character, a missing character, an extra character or a transposition.

Correct Word Spelling *Command* C-$, Alt $

This command looks up the previous or current word in the dictionary, and attempts to correct the spelling if it is misspelled. There are four possible results of this action:

1. The message "`Found it.`" is displayed in the echo area. This means that the word was found in the dictionary exactly as given.
2. The message "`Found it because of` *word*`.`" is displayed, where *word* is some other word with the same root, but a different ending. The word is no less correct than if the first message is given, but an additional piece of useless information is supplied to make you feel like you are using a computer.
3. The message "`Word not found.`" is displayed. Either the word is not in the dictionary or is so badly mangled that the correct spelling cannot be found. If this happens, it is worth trying some alternate spellings, as one of them is quite likely close enough to be found.
4. The prompt "`Correction choice:`" appears in the echo area and a list of numbers and words appears in a pop-up window. Typing a number selects the corresponding correction, which replaces the erroneous word preserving case, as though by Query Replace. Typing anything else rejects all the choices.

Correct Buffer Spelling *Command*

This command scans the entire buffer looking for misspelled words and offering to correct them. A window into the Spell Corrections buffer is placed on the screen, and a log of any actions taken is maintained in that buffer. When an unknown word is found, a single-character command is prompted for:

A	Ignore this word. If it is encountered again, then the prompting is repeated.
I	Insert this word in the dictionary.
C	Choose one of the corrections displayed in the Spell Corrections window by specifying the correction number. If the same misspelling is encountered again, then the correction will be done automatically, leaving a note in the log window.
R	Prompt for a word to use instead of the offending one, remembering the correction the same way that **C** does.
C-R	Go into a recursive edit at the current position, and resume checking when the recursive edit is exited.

After this command completes it deletes the log window, but leaves the buffer around for future reference.

Add Word to Spelling Dictionary *Command* **C-X $**

This command adds the previous or current word to the spelling dictionary.

Augment Spelling Dictionary *Command*

This command adds some words from a file to the spelling dictionary. The format of the file is a list of words, one on each line.

Append to Spelling Dictionary *Command*

This command appends incremental dictionary insertions to a file. Any words added to the dictionary since the last time this was done will be appended to the file. All commands which add words to the dictionary except Augment Spelling Dictionary add their insertions to this list.

4.4.1 Auto Spell Mode

Auto Spell Mode checks the spelling of each word as it is typed. When an unknown word is typed the user is notified and allowed to take a number of actions to correct the word.

Auto Spell Mode *Command*

This command turns Spell mode on or off in the current buffer.

Check Word Spelling *Command* `Spell: word-delimiters`

This command checks the spelling of the word before the point, doing nothing if the word is in the dictionary. If the word is misspelled but has a known correction then the correction is made. If there is no correction then a message is displayed in the echo area. An unknown word detected by this command may be corrected using the Correct Last Misspelled Word command. These actions are performed in addition to the normal action for the key bound.

Check Word Spelling Beep *Hemlock Variable* T

If this variable is true, then Check Word Spelling will beep when an unknown word is found.

Correct Last Misspelled Word *Command* `Spell: C-:`

This command moves the cursor to after the last misspelled word detected by the Check Word Spelling command and then prompts for a single character command:

C	Offer a choice of possible corrections.
I	Insert the word in the dictionary.

R	Replace the word with another.
C-H, Backspace, Delete	Skip this word and try again on the next most recently misspelled word.
C-R	Enter a recursive edit at the word, exiting the command when the recursive edit is exited.
Alt	Exit and forget about this word.

As in Correct Buffer Spelling, the **C** and **R** commands add the correction to the known corrections.

Chapter Five
Managing Large Systems

Hemlock provides two tools which help to manage large systems:

1. File groups, which provide several commands that operate on all the files in a possibly large collection, instead of merely on a single buffer.
2. A source comparison facility with semi-automatic merging, which can be used to compare and merge divergent versions of a source file.

5.1 File Groups

A file group is a set of files, upon which various editing operations can be performed. The files in a group are specified by a file, the syntax of which is compatible with an Update storage command file, and is described here:

- Any line which begins with one "@" is ignored.
- Any line which does not begin with an "@" is the name of a file in the group.
- A line which begins with "@@" specifies another file having this syntax, which is recursively examined to find more files in the group.

Although any number of file groups may be read into Hemlock, there is only one *active group*, which is the file group implicitly used by all of the file group commands.

Select Group *Command*

This command prompts for the name of a file group to make the active group. If the name entered is not the name of a group whose definition has been read, then the user is prompted for the name of a file to read the group definition from. The name of the default pathname is the name of the group, and the type is "Upd".

Group Query Replace *Command*

This command prompts for a target and replacement string, and then does an interactive string replace on each file in the active group. Each file is read in as though by **Find File** then processed as though **Query Replace** had been given the specified target and replacement strings.

Group Replace *Command*

This is like Group Query Replace except that it does a non-interactive replacement, similar to Replace String.

Group Search *Command*

This command prompts for a string, and then searches for it in each file in the active group. When an occurrence is found, the user is prompted for a single-character command to indicate what action to take. The following commands are defined:

Space, Y	Continue searching for the next occurrence of the string.
Delete, Backspace, N	Continue the search at the beginning of the next file, skipping the remainder of the current file.
Alt	Exit Group Search.
C-R	Go into a recursive edit at the current location, and continue the search when it is exited.

5.2 SrcCom

These two commands can be used to find exactly how the text in two buffers differs, and to generate a new version that combines features of both versions. There are some known bugs in these commands, but they are still very useful.

Compare Buffers *Command*

This command prompts for three buffers, and then does a buffer comparison. The first two buffers must exist, as they are the buffers to be compared. The last buffer, which is created if it does not exist, is the buffer to which output is directed. The output buffer is selected during the comparison so that its progress can be monitored. The output is self-explanatory. There are various variables that control exactly how the comparison is done.

Merge Buffers *Command*

This command functions in a very similar fashion to Compare Buffers, the difference being that a version which is a combination of the two buffers compared is generated in the output buffer. Text that is identical in the two comparison buffers is copied unchanged to the output buffer. When a difference is encountered, the two differing versions are displayed in the output buffer, and the user is prompted for an action to take. The following single-character commands are defined:

1	Use the first version of the text.
2	Use the second version.
B	Insert both versions, distinctively marked with the string "**** MERGE LOSSAGE ****". This is useful if the change that needs to be made is too complex to be done conveniently at this point, or it is unclear which version is correct. After the merge is complete, this string may be easily found with a search command.
C-R	Do a recursive edit, and ask again when the edit is exited.

Source Compare Ignore Case *Hemlock Variable* `nil`

If this variable is true, Compare Buffers and Merge Buffers will do comparisons case-insensitively. Turning this on will slow down these commands significantly.

Source Compare Ignore Extra Newlines *Hemlock Variable* `t`

If this variable is true, Compare Buffers and Merge Buffers will treat all groups of newlines as if they were a single newline.

Source Compare Number of Lines *Hemlock Variable* 3

This variable controls the number of lines Compare Buffers and Merge Buffers will compare when resynchronizing after a difference has been encountered.

Chapter Six
Editing Lisp

Hemlock provides a large number of powerful commands for editing LISP code. It is possible for a text editor to provide a much higher level of support for editing LISP than ordinary programming languages, since its syntax is much simpler.

Currently the LISP specific commands do not parse LISP code totally correctly, the most serious flaw being that they do not understand comments or quotation mechanisms. This is not usually a problem unless there is a parenthesis in a command, quoted string, character literal or symbol. Another limitation is that all LISP commands abort and signal an error if they cross a defun boundary when parsing. This is done to speed the detection of errors, so that it is not necessary to parse all the way back to the beginning of the buffer if, for example, an extra close parenthesis is typed.

6.1 Lisp Mode

Lisp mode is a major mode used for editing LISP code. It has a number of mode-local key bindings that shadow global key bindings.

Lisp Mode *Command*

Set the major mode of the current buffer to Lisp.

6.2 Form Manipulation

These commands manipulate LISP forms, the printed representations of LISP objects. A form is either an expression balanced with respect to parentheses or an atom such as a symbol or string.

Forward Form *Command* **C-Z f**
Backward Form *Command* **C-Z b**

Forward Form moves to the end of the current or next form, while Backward Form moves to the beginning of the current or previous form. A prefix argument is treated as a repeat count.

Forward Kill Form *Command* **C-Z k**
Backward Kill Form *Command* **C-Z Delete, C-Z Backspace**

Forward Kill Form kills text from the point to the end of the current form. If at the end of a list, but inside the close parenthesis, then kill the close parenthesis. Backward Kill Form is the same, except it goes in the other direction. A prefix argument is treated as a repeat count.

Mark Form *Command* C-Z @
This command sets the mark at the end of the current or next form.

Transpose Forms *Command* C-Z t
This command transposes the forms before and after the point and moves forward. A prefix argument is treated as a repeat count. If the prefix argument is negative, then the point is moved backward after the transposition is done, reversing the effect of the equivalent positive argument.

Insert () *Command* Alt (
This command inserts an open and a close parenthesis, leaving the point inside the open parenthesis. If a prefix argument is supplied, then the close parenthesis is put at the end of the form that many forms from the point.

6.3 List Manipulation

List commands are similar to form commands, but the only pay attention to lists, ignoring any atomic objects that may appear. These commands are useful because they can skip over many symbols, and move up and down in the list structure.

Forward List *Command* C-Z n
Backward List *Command* C-Z p
Forward List moves the point to immediately after the end of the next list at the current level of list structure. If there is not another list at the current level, then it moves up to past the end of the containing list. Backward List is identical, except that it moves backward and leaves the point at the beginning of the list. The prefix argument is used as a repeat count.

Forward Up List *Command* C-Z)
Backward Up List *Command* C-Z u
Forward Up List moves to the after the end of the enclosing list. Backward Up List moves to the beginning. The prefix argument is used as a repeat count.

Down List *Command* C-Z d

This command moves to just after the beginning of the next list. The prefix argument is used as a repeat count.

Extract List *Command* C-Z x

This command "extracts" the current list from the list which contains it. The outer list is deleted, leaving behind the current list. The entire affected area is pushed on the kill ring, so that this possibly catastrophic operation can be undone. The prefix argument is used as a repeat count.

6.4 Defun Manipulation

A *defun* is a list whose open parenthesis is against the left margin. It is called this because an occurrence of the `defun` top level form usually satisfies this definition, but they will work on any top level form, such as a `defstruct` or `defmacro`.

End of Defun *Command* C-Z e, C-Z]
Beginning of Defun *Command* C-Z a, C-Z [

End of Defun moves to the end of the current or next defun. Beginning of Defun moves to the beginning of the current or previous defun. End of Defun will not work if the parentheses are not balanced.

Mark Defun *Command* C-Z h

This command puts the point at the beginning and the mark at the end of the current or next defun.

6.5 Indentation

One of the most important features provided by Lisp mode is automatic indentation of LISP code, since unindented LISP is unreadable, poorly indented LISP is ugly, and inconsistently indented LISP is subtly misleading.

Indent for Lisp *Command* `Lisp`: Tab, C-I

This command indents a line of LISP according to the standard indentation conventions. If the point is within the indentation, then it is moved to the first non-blank character, otherwise it is left where it is. If a prefix argument is supplied, then that many lines below the current line are indented.

Indent Form *Command* C-Z q
This command indents all the lines in the current form, leaving the point unmoved.

Lisp Indent Region *Command* C-Z \
This command indents all of the lines in the region.

Defindent *Command* C-Z #
This command prompts for the number of special arguments to associate with the symbol at the beginning of the current or containing list.

Lisp New Line *Command* Lisp: Linefeed
This command goes to a new line, as though by New Line, and then inserts the correct amount of indentation for LISP code.

Move Over) *Command* Alt)
This command moves past the next close parenthesis and then does what Lisp New Line does.

6.6 Parenthesis Matching

Another very important facility provided by Lisp mode is *parenthesis matching.* Whenever a close parenthesis is inserted in Lisp mode, the matching open parenthesis is indicated.

Lisp Insert) *Command* Lisp:)
Paren Pause Period *Hemlock Variable* 0.5
This command inserts a close parenthesis and then attempts to display the matching open parenthesis by placing the cursor on top of it for Paren Pause Period seconds. If there is no matching parenthesis then the screen is flashed. If the matching parenthesis is off the top of the screen, then the line on which it appears is displayed in the echo area.

Chapter Seven
Interacting With Lisp

There are a number of facilities for querying and modifying the LISP environment while in Hemlock.

7.1 The Current Package

The value of `*package*` during the execution of these commands is the same as the package that `ed` was called in, unless a `Package` file option has been specified for the current buffer, in which case it is the specified package. Setting `*package*` using one of these commands will cause the current value to be permanently changed, thus setting `*package*` in a buffer with a local package will only change the local package.

Set Buffer Package *Command*

This command prompts for the name of a package to make the local package in the current buffer.

7.2 Input and Output

Reading from a terminal stream (`*standard-input*`, `*terminal-io*`) while in Hemlock is not recommended, since it doesn't work very well. The reading will be done from the `"lisp"` window even if the prompt is displayed in Hemlock.

Output to `*standard-output*` is redirected by Top-Level mode and the compilation commands, and is thus treated in a somewhat sensible fashion. Any output not redirected will be displayed in the `"lisp"` window.

7.3 Compiling and Evaluating Lisp Code

These commands can greatly speed up the edit/debug cycle, since they enable incremental reevaluation or recompilation of changed code, avoiding the need to compile and load an entire file.

Evaluate Expression *Command* C-Alt, Alt Alt

This command prompts for an expression and prints the result of its evaluation in the echo area.

Load File *Command*
Load Pathname Defaults *Hemlock Variable* `nil`

This command prompts for a file and calls the `load` function with it. Load Pathname Defaults contains the default pathname for this command. This variable is set to the file loaded; if it is `nil` then there is no default.

Evaluate Defun *Command* C-X C-E
Evaluate Region *Command*
Evaluate Buffer *Command*

These commands evaluate text out of the current buffer, reading the current defun, the region and the entire buffer, respectively.

Compile Defun *Command* C-X C-C
Compile Region *Command*

These commands recompile the text in the current defun and the region, respectively. Compiler information output is directed to a pop-up window.

Compile File *Command* C-X c

This command saves the current buffer if it is modified, and then calls `compile-file` on the file. All compiler information is placed in the Compiler Warnings buffer, a window being made into this buffer if there is not one already. Since there is a complete log of output in the Compiler Warnings buffer, the creation of the normal error output (`".err"`) file is inhibited. Note that unlike the other compiling and evaluating commands, this does not have the effect of placing the definitions in the environment, to do so, the resulting output (`".sfasl"`) file must be loaded. If a prefix argument is specified, then the user is prompted for a file to compile instead of the one in the current buffer.

7.4 Querying the Environment

These commands are useful for obtaining various random information from the LISP environment.

Describe Function Call *Command* C-Z M-A
Describe Symbol *Command* C-Z M-S

Describe Function Call describes the symbol found at the head of the currently enclosing list. Describe Symbol describes the symbol at or before the point. These commands are primarily useful for finding the documentation for functions and variables.

Lisp Describe *Command* **Help t**

This command prompts for an expression, evaluates it, and describes the result of the evaluation.

Room *Command*

Call the `room` function, which displays information about allocated storage, directing output to a pop-up window.

7.5 Top-Level Mode

Top-Level mode is a minor mode which is used to have the effect of a read eval print loop in a Hemlock buffer. **Warning:** it is dangerous to interrupt Hemlock while it is inserting output into a buffer, as the buffer structure may be damaged. If this happens there is no way to recover other than to start a new LISP, although undamaged buffers may be saved using the `save-all-buffers` function.

Top-Level Mode *Command*

this command turns on Top-Level mode in the current buffer, and sets the major mode to Lisp, if it is not already. This is normally done in a buffer specially created for the purpose, since the output inserted into the buffer tends to trash it. If Top-Level mode is already on, then it is turned off.

Top-Level Eval *Command* **`Top-Level`: Return**

This command evaluates all the forms between the end of the last output and the end of the buffer, inserting the results of their evaluation in the buffer. If the point is before the position of the prompt then the form which ends on the current line is inserted at the end of the buffer and evaluated. This command will complain if there is not a complete form, so to insert line breaks in the middle of a form, use **Linefeed.**

Kill Top-Level Input *Command* **`Top-Level`: Oops**

This command kills input that would be read by Top-Level Eval.

Abort Top-Level Input *Command* **`Top-Level`: C-Oops**

This command moves the the end of the buffer and prompts, ignoring any input already typed in.

Next Top-Level Input *Command* `Top-Level: M-N`
Previous Top-Level Input *Command* `Top-Level: M-P`

A history of inputs to Top-Level Eval is maintained. These commands step forward and backward in the history, inserting the current entry in the buffer. The prefix argument is used as a repeat count.

Top-Level Beginning of Line *Command* `Top-Level: C-A`

This command is identical to Beginning of Line unless there is no prefix argument and the point is on the same line as the prompt; then it moves to immediately after the prompt.

7.6 Error Handling

When an error happens inside of Hemlock, Hemlock will trap the error and display the error message in the echo area, possibly along with the "`Internal error:`" prefix. If you want to debug the error, type ?. This causes the prompt "`Debug:`" to appear in the echo area. The following commands are recognized:

D	Enter a break-loop so that you can use the LISP debugger. Proceeding with "`$p`" will reenter Hemlock, and give the "`Debug:`" prompt again.
B	Show a stack backtrace in a pop-up window.
Q, Alt	Quit from this error to the nearest command loop.

Chapter Eight
Other Languages

Currently only one language other than LISP is supported, Pascal. This support, although not very great, is comparable to that of many non-extensible editors.

Pascal Mode *Command*

This command sets the current buffer's major mode to Pascal. This mode has a crude indentation facility, and borrows parenthesis matching from LISP.

Generic Indent *Command* **Pascal: Tab, C-I**

This command indents the current line the same distance as the previous non-blank line.

Generic New Line and Indent *Command* **Pascal: Linefeed**

This command goes to a new line like New Line, and then indents like Generic Indent.

Chapter Nine
Simple Customization

Hemlock can customized and extended to a very large degree, but in order to do much of this a knowledge of LISP is required. These advanced aspects of customization are discussed in the *Hemlock Command Implementor's Manual*, while simpler methods of customization are discussed here.

9.1 Keyboard Macros

Keyboard macros provide a facility to turn a sequence of commands into one command.

Define Keyboard Macro *Command* **C-X (**
End Keyboard Macro *Command* **C-X)**

Define Keyboard Macro starts the definition of a keyboard macro. The commands which are invoked up until End Keyboard Macro is invoked become the definition for the keyboard macro, thus replaying the keyboard macro is synonymous with invoking that sequence of commands.

Last Keyboard Macro *Command* **C-X e**

This command is the keyboard macro most recently defined; invoking it will replay the keyboard macro. The prefix argument is used as a repeat count.

Keyboard Macro Query *Command* **C-X q**

This command is used to conditionalize the execution of a keyboard macro. When invoked during the definition of a macro, it does nothing, but when the macro is replayed it prompts the user for a single-character command to indicate the action to be taken. The following commands are defined:

Alt	Exit all repetitions of this keyboard macro. More than one may have been specified using a prefix argument.
Space, Y	Proceed with the execution of the keyboard macro.
Delete, Backspace, N	Skip the remainder of the keyboard macro and go on to the next repetition, if any.
!	Do all remaining repetitions of the keyboard macro without prompting.

Complete this repetition of the macro and then exit without doing any of the remaining repetitions.

C-R Do a recursive edit, then prompt again.

Name Keyboard Macro *Command*

This command prompts for the name of a command, and then makes the definition for that command the same as Last Keyboard Macro's current definition. The command which results is not clobbered when another keyboard macro is defined, so it is possible to keep several keyboard macros around at once. The resulting command may also be bound to a key using Bind Key, in the same way any other command is.

Many keyboard macros are not for customization, but rather for one-shot use, a typical being performing some operation on each line of a file. To add "`del `" to the beginning and "`.*`" to the end of every line in in a buffer, one could do this:

C-X (d e l Space C-E . * C-N C-A C-X) C-U 9 9 9 C-X e

First a keyboard macro is defined which performs the desired operation on one line, and then the keyboard macro is invoked with a large prefix argument. The keyboard macro will not actually execute that many times; when the end of the buffer is reached the C-N will get an error and abort the execution.

9.2 Binding Keys

Bind Key *Command*

This command prompts for a command, a key and a kind of binding to make, and then make the specified binding. The following kinds of bindings are allowed:

buffer Prompts for a buffer, and then makes a key binding which is only present when that buffer is the current buffer.

mode Prompts for the name of a mode, and then make a key binding which is only in present when that mode is active in the current buffer.

global A global key binding is made, which is in effect when there is no applicable mode or buffer key binding. This is the default.

Delete Key Binding *Command*

This command prompts for a key binding the same way that Bind Key does, and makes the specified binding go away.

9.3 Hemlock Variables

A number of commands user Hemlock variables as flags to control their behavior.

Set Variable *Command*

This command prompts for the name of a Hemlock variables and an expression, then sets the current value of the variable to the result of the evaluation of the expression.

9.4 Init Files

Hemlock customizations are normally put in the main LISP initialization file, `"init.slisp"`, or when compiled `"init.sfasl"`. The contents of the init file must be LISP code, but there is a fairly straightforward correspondence between the basic customization commands and the equivalent LISP code. Rather than describe these functions in depth here, a brief example will be given:

```
;;; -*- Mode: Lisp; Package: Hemlock -*-

;;; It is necessary to specify that the customizations go in
;;; the hemlock package.
(in-package 'hemlock)

;;; Bind Kill Previous Word to M-H.
(bind-key "Kill Previous Word" '#(#\m-h))
;;;
;;; Bind Extract List to C-Z ? when in Lisp mode.
(bind-key "Extract List" '#(#\c-z #\?) :mode "Lisp")

;;; Make C-W globally unbound.
(delete-key-binding '#(#\c-w))

;;; Make string searches case-sensitive.
(setv default-search-kind :string-sensitive)
;;;
;;; Make query replace replace strings literally.
(setv case-replace nil)
```

For a detailed description of what these functions do, see the *Hemlock Command Implementor's Manual.*