Hemlock User's Manual

Rob MacLachlan

September 1987

CMU-CS-87-158

Abstract

This document describes the Hemlock text editor, as of version M2.3. Hemlock is a customizable, extensible text editor whose initial command set closely resembles that of ITS/TOPS-20 Emacs. Hemlock is written in CMU COMMON LISP and has been ported to other implementations.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Space and Naval Warfare Systems Command under proposed contract N00039-87-C-0251.

Table of Contents

Index

Chapter 1

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 current focus of editing activity. Text typed in by the user 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 seem a bit curious at first, but it is necessary 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 visible indication of the current focus of attention: a rectangular blotch under X windows, or the hardware cursor on a terminal. The cursor is usually displayed on the character which is immediately after the point, but it may be displayed in other places. 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. Under X windows the cursor is only displayed when when Hemlock is waiting for input.

1.2. Notation

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

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 the

[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.

keyboard places limitations on what characters can actually be typed, Hemlock can understand arbitrary combinations of four bits: control, meta, super and hyper. The bits in a character are represented by prefixing the character with combinations of **C-**, **M-**, **S-** and **H-**. For example, **a** with both the control and meta bits set is written **C-M-a**.

Bits are totally independent modifiers of the character, so (unlike in ASCII) it is possible to have both uppercase and lowercase control characters. For example, **C-a** and **C-A** may mean different things. Generally, Hemlock ignores the case of alphabetic characters. When case is unimportant, the lowercase character will be used. If a character must be shifted, then it will be represented in uppercase.

Some characters such as **Home**, **Return** and **Delete** 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.

See also sections 1.8 and 1.7.

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 (bound to **C-M-q**, **C-'**) [*Command*]

This command's name is Sample Command, *and it is bound to* **C-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.

1.2.3. Hemlock Variables

Hemlock variables supply a simple customization mechanism by permitting commands to be parameterized. For details see page 64.

Sample Variable (initial value `36`) [Hemlock *Variable*]

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. See section 1.2.1 for a discussion what characters Hemlock recognizes and sections 1.8 and 1.7 to find out how to type characters on different devices. 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 section 1.12), and can usually find out what options are available by typing **C-_** or **Home** (see section 1.10).

The user can easily alter old key bindings or bind commands not previously bound (see section 10.2).

In addition to the key bindings explicitly listed with each command, there are a number of implicit key bindings created by using key translations[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 usually not significant in key bindings, since most uppercase characters have a translation to the corresponding lowercase character. Case insensitive bindings are always represented as lowercase (e.g. **C-a**). Case sensitive bindings are represented using the appropriate case (e.g. **C-M-a** or **C-M-A**).
- The bit-prefix characters **Escape** and **C-z** may be used in key bindings to convert the following character to a meta or control-meta character. For example, **C-x Escape b** may be used instead of **C-x M-b** and **C-z u** may be used instead of **C-M-u**. This allows Hemlock to be used with keyboards that don't have a meta key.

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 (bound to **M-x**) [*Command*]

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 **Escape** and **Space**, and a list of possible completions is given by **Home** or **C-_**.

1.4. 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.

---

[2]Key translations are documented in the *Hemlock Command Implementor's Manual*.

- 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 (bound to all meta digits) [*Command*]

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

Negative Argument (bound to **M--**) [*Command*]

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

Universal Argument (bound to **C-u**) [*Command*]

Universal Argument Default [Hemlock *Variable*]

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. **M-4 M-2 C-u** sets the prefix argument to one hundred and sixty-eight. **C-u M-0** sets the prefix argument to forty.

Universal Argument Default determines the default value and multiplier for the Universal Argument command.

1.5. 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; Hemlock is 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 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 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 may not be the case when key bindings conflict.

Modes can be envisioned as switches, the major mode corresponding to 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.

Fundamental Mode [*Command*]

This command puts the current buffer into Fundamental mode. Fundamental mode is the most basic major mode: it's the next best thing to no mode at all.

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 **Delete**, **Backspace** or **n** is typed, then command continues, but does not remove the output window. This allows the output from the command to be kept around for reference even after the command completes. If any other character is typed, then the pop-up window will still go away, but the character will be re-read as well, and thus will 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** or **y** will go on to the next window full, while **Delete**, **Backspace** or **n** aborts the remaining output. If any other character is typed, the remaining output will be aborted, but the window will not be removed.

Once you exit more-more in a way that retains the pop-up window, the only way to get rid of it is to use a screen manager command such as iconify.

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 in the last column of each wrapped line. Currently, the wrap character is always a exclamation point (`!`). 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.

1.6.3. The Modeline

The modeline is the line displayed 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)  /usr/slisp/hemlock/user.mss
```

This tells us that the file associated with this buffer is "`/usr/slisp/hemlock/user.mss`" 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:

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

In this case, the buffer is named Silly and is in Lisp mode.

If the buffer displayed in a window has been modified since the last time it was read from or save to a file, then an asterisk (`*`) will be displayed after the file or buffer name:

```
Hemlock  (Fundamental Fill)  /usr/slisp/hemlock/user.mss  *
```

This serves as a reminder that the buffer should be saved eventually.

1.7. Use With X Windows

It is preferable to use Hemlock on a workstation with a bitmap display and a window manager, since Hemlock makes good use of the window manager and non-ASCII input devices such as the mouse and modifier keys. This section deals with using Hemlock under X windows, which is currently the only supported window manager.

1.7.1. Event Translation

Each X key event is translated into a single LISP character. The control and meta (**Alt** on the RT) modifiers are directly translated to the LISP control and meta bits. The `char-code` for the character is determined by a combination of the X scan code and the shift (or caps-lock) modifiers.

If a key is shifted, and there is an obvious ASCII translation, then the code for the shift will be used. For example, the shift of **3** is **#**. The shift of keys that don't have a distinct shifted character are translated to that character with the super bit on. For example, the shift of **F1** is **S-F1**. Shifting doesn't affect **Tab**, **Space**, **Backspace**, **Delete**, **Return** and **Linefeed**.

Numeric keypad keys that duplicate normal keys are translated to the normal character with the super bit on. For example, **9** on the keypad is **S-9**. Shifting these keys has no effect.

Note that with the stupid two-button mouse on the IBM RT PC, the only way to to send **Middledown** is to press both the left and right buttons simultaneously. For this reason, the commands bound to the middle button are also bound to shift of the left button, i.e. **S-Leftdown**.

1.7.2. Cut Buffer Commands

These commands allow the X cut buffer to be used from Hemlock . Although Hemlock can cut arbitrarily large regions, a bug in the standard version 10 xterm prevents large regions from pasted into an xterm window.

Region to Cut Buffer (bound to **M-Insert**) *[Command]*
Insert Cut Buffer (bound to **Insert**) *[Command]*

These commands manipulate the X cut buffer. Region to Cut Buffer puts the text in the region into the cut buffer. Insert Cut Buffer inserts the contents of the cut buffer at the point.

1.7.3. Redisplay and Screen Management

These variables control a number of the characteristics of Hemlock bitmap screen management.

Bell Style (initial value `:border-flash`) [Hemlock *Variable*]
Beep Border Width (initial value `20`) [Hemlock *Variable*]

Bell Style determines what beeps do in Hemlock. Acceptable values are `:border-flash`, `:feep`, `:border-flash-and-feep`, `:flash`, `:flash-and-feep`, and `nil` (do nothing).

Beep Border Width is the width in pixels of the border flashed by border flash beep styles.

Reverse Video (initial value `nil`) [Hemlock *Variable*]

If this variable is true, then Hemlock paints white on black in window bodies, black on white in modelines.

Set Window Autoraise (initial value `:echo-only`) [Hemlock *Variable*]

When true, changing the current window will automatically raise the new current window. If the value is `:echo-only`, then only the echo area window will be raised automatically upon becoming current.

Initial Window Geometry (initial value `""`) [Hemlock *Variable*]
Initial Window Default Geometry (initial value `"80x24+1+1"`) [Hemlock *Variable*]
Default Default Geometry (initial value `"80x24+1+1"`) [Hemlock *Variable*]

These variables hold window geometry specifications that are used when creating windows interactively.

Initial Window Geometry is used to create the initial window. When this is an empty string, the user is prompted for the size and position of the initial window. If a complete geometry specification is given, then the window will be created without prompting.

The other variables determine the default geometry for interactive window creation. Initial Window Default Geometry is used as the default for the initial Hemlock window. Default Default Geometry is used as the default geometry when making a window with New Window (page 30).

The format of an X geometry specification is the following:

`[`*width*`] [x`*height*`] [{+|-}`*xoff*`[{+|-}`*yoff*`]]`

width and *height* are in character units, and *xoff* and *yoff* are in pixels. *xoff* and *yoff* designate offsets from a corner of the screen to a corresponding corner of the window:

+xoff+yoff	upper left to upper left
-xoff+yoff	upper right to upper right
+xoff-yoff	lower left to lower left
-xoff-yoff	lower right to lower right

Cursor Bitmap File (initial value `"/usr/misc/.lisp/lib/hemlock.cursor"`) [Hemlock *Variable*]

This variable determines where the mouse cursor bitmap is read from when Hemlock starts up. The mask is found by merging this name with `".mask"`. This has to be a full pathname for the C routine.

1.8. Use With Terminals

Hemlock can also be used with ASCII terminals and terminal emulators. Capabilities that depend on X windows (such as mouse commands) are not available, but nearly everything else can be done.

1.8.1. Terminal Input

The most important limitation of a terminal is in the input capabilities. On a workstation with function keys and independent control, meta and shift modifiers, it is possible to type 800 or so distinct single keystrokes. Although by default, Hemlock uses only a fraction of these combinations, there are still many more than the 128 characters available in ASCII.

On a terminal, Hemlock attempts to translate ASCII "control characters" in into the most useful full character:

- On a terminal, control doesn't compose with shift, so it isn't possible to make case distinctions in control characters. Hemlock always translates `control-`*letter* into a lowercase control character.
- On a terminal, some of the named keys generate the same ASCII character as control-key combinations. For example, `return` and `control-m` are identical. Hemlock translates such ambiguous characters to the named character in favor of the control character, so both `return` and `control-m` will translate to **Return**.

Since terminals have no meta key, the **Escape** and **C-Z** bit-prefix characters must be used to invoke commands bound to control or control-meta characters.

When running Hemlock from a terminal `^\` is the interrupt character. Typing this will place you in the Lisp debugger.

When using a terminal, pop-up output windows cannot be retained after the completion of the command.

1.8.2. Terminal Redisplay

Redisplay is substantially different on a terminal. Different algorithms are used, and there are different parameters that control how redisplay and screen management are done.

Terminal redisplay uses the Unix termcap database to find out how to use a terminal. Hemlock can be used with terminals that lack insert/delete line/character capability. On terminal emulators that implement these operations very inefficiently (such as xterm), it is desirable to use a termcap entry with the losing capabilities deleted.

Scroll Redraw Ratio (initial value `nil`) [Hemlock *Variable*]

This is a ratio of "inserted" lines to the size of a window. When this ratio is exceeded, insert/delete line terminal optimization is aborted, and every altered line is simply redrawn as efficiently as possible. For example, setting this to 1/4 will cause scrolling commands to redraw the entire window instead of moving the bottom two lines of the window to the top (typically 3/4 of the window is being deleted upward and inserted downward, hence a redraw); however, commands like New Line and Open Line will still work efficiently, inserting a line and moving the rest of the window's text downward.

1.9. The Echo Area

The echo area 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.

When a command needs some information form the user, it requests it by displaying a *prompt* in the echo area.

Here is a typical prompt:

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

The general format of a prompt is a one or two word description of the input requested, possibly followed by a *default* in brackets. The default is a standard response to the prompt that is automatically supplied if you type return without giving any other input.

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 **Escape**, 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 **Escape** so as to distinguish entering an old keyword from making a new keyword which is a prefix of an old one.

file — The response is the name of a file, which may be required to exist. Unlike for other prompts, the default has some effect even when input has been given: the default is *merged* with the input filename. Filename merging is described on page 28. **Escape** can be used to complete the input to a file parse.

string — The response is a string which must satisfy some property, such as being the name of an existing 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, then beep and give the user a chance to fix the problem.

Home, C-_ — 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.

Escape — Attempt to complete the input to a keyword parse as far as possible, beeping 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 beeping a lot. For example, Forward Word can be invoked as an extended command by typing **M-X f o Space w Return**.

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

C-p — Retrieve the text of the last string input from a history of echo area inputs. 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.

1.10. Online Help

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

Help (bound to **Home, C-_**) [*Command*]

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.
Home, C-_, ?, h	List all of the options and what they do.

What Lossage (bound to **Home l, C-_ l**) [*Command*]
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 (bound to **Home w, C-_ w**) [*Command*]
This command prompts for the name of a command and displays its key bindings in a pop-up window. If a key binding is not global, the environment in which it available is displayed.

Apropos (bound to **Home a, C-_ a**) [*Command*]
This command prints brief documentation for all commands, variables and character attributes whose names contain a specified string. The bindings of commands and values of variables are printed with the documentation.

Describe Command (bound to **Home d, C-_ d**) [*Command*]
This command prompts for a command and prints its full documentation.

Describe Key (bound to **Home c, C-_ c, M-?**) [*Command*]
This command prints full documentation for the command which is bound to the specified key in the current environment.

Generic Describe (bound to **Home g, C-_ g**) [*Command*]
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.

1.11. Entering and Exiting

Hemlock is entered by using the COMMON 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. The **-edit** command-line switch may also be used to enter Hemlock: see page 61.

ed may optionally be given a file name or a symbol argument. 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 adding "**Edit** " to the beginning of the symbol's name.

Exit Hemlock (bound to **C-c, C-x C-z**) [*Command*]

Pause Hemlock [*Command*]

Exit Hemlock 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. At any time after exiting you may reenter by typing `(ed)` to LISP without losing anything. Before you quit from LISP using `(quit)`, you should save any modified files that you want to be saved.

Pause Hemlock is similar, but it suspends the LISP process and returns control to the shell. When the process is resumed, it will still be running Hemlock.

1.12. 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 **Home** or **C-_**.
- 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-x C-z** 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. **C-x C-m** may be used to save all modified buffers which have associated files. 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, make sure you have saved all your changes. **C-u C-x C-b** will display a list of all modified buffers. If you exit using **C-x M-z**, then modified buffers with associated files will automatically be written out.
- If the screen changes unexpectedly, you may have accidentally typed an incorrect command. Use **Home 1** to see what it was. If you are not familiar with the command, use **Home c** to see what it 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**. The Undo command is also useful for recovering from this sort of problem.

Region Query Size (initial value 30) [Hemlock *Variable*]

Various commands ask for confirmation before modifying a region containing more than this number of lines. If this is `nil`, then don't ask, no matter how large the region in.

Undo [*Command*]

This command may be used to undo the last major modification. Killing commands and certain other commands save information about the modifications they make so that accidental uses may be recovered from. When used, this command displays the name of the operation to be undone and asks for confirmation. If the affected text has been modified between the Undo and the command to be undone, then the results may be somewhat incorrect, but are probably still useful. Often Undo itself can be undone.

1.13. Recursive Edits

Some sophisticated commands, such as Query Replace, can place you in a *recursive edit.* A recursive edit is simply a recursive invocation of Hemlock done within a command. A recursive edit is useful because it allows arbitrary editing to be done during the execution of a command without losing any state that the command might have. When 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 (bound to **C-M-z**) [*Command*]

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

Abort Recursive Edit (bound to **C-]**) [*Command*]

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 and return the string "`Recursive edit aborted.`".

1.14. User Errors

When in the course of editing, Hemlock is unable to do what it thinks you want to do, then it brings this to your attention by a beep or a screen flash (possibly accompanied by an explanatory echo area message such as "`No next line.`".) Although the exact attention-getting mechanism may vary on the output device and variable settings, this is always called *beeping*.

Whatever the circumstances, you had best try something else since Hemlock, being far more stupid than you, is far more stubborn. Hemlock is an extensible editor, so it is always possible to change the command that complained to do what you wanted it to do.

1.15. Internal Errors

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

```
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 your 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 save any changes with `save-all-buffers` 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 debugger ("`1] `") prompt and answer the questions it asks. Since this function will prompt in the "`Lisp`" window, it isn't very useful when called inside of

Hemlock.

Chapter 2

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 (bound to **C-f, Rightarrow**) [*Command*]
Backward Character (bound to **C-b, Leftarrow**) [*Command*]

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 (bound to **M-f**) [*Command*]
Backward Word (bound to **M-b**) [*Command*]

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 that 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 (bound to **C-n, Downarrow**) [*Command*]
Previous Line (bound to **C-p, Uparrow**) [*Command*]

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 (bound to **C-e**) [*Command*]
Beginning of Line (bound to **C-a**) [*Command*]

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 (bound to **C-v**) [*Command*]
Scroll Window Up (bound to **M-v**) [*Command*]

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 Scroll Window Down, except that it moves backwards.

Scroll Overlap (initial value 2) [Hemlock *Variable*]

This variable is used by Scroll Window Down and Scroll Window Up to determine the number of lines by which the new and old screen should overlap.

End of Buffer (bound to **M-<**) [*Command*]
Beginning of Buffer (bound to **M->**) [*Command*]

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 mark stack (see page 16).

Top of Window (bound to **M-,**) [*Command*]
Bottom of Window (bound to **M-.**) [*Command*]

Top of Window moves the point to the beginning of the first line displayed in the current window. Bottom of Window moves to the beginning of the last line displayed.

2.2. The Mark and The Region

Each buffer has a distinguished position known as the *mark*. The mark initially points to 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 is 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.

By definition, the region 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.

Exchange Point and Mark (bound to **C-x C-x**) [*Command*]

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 (bound to **C-x h**) [*Command*]

This command sets the region around the whole buffer, with the point at the beginning and the mark at the end. 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 the stack twice will restore it.

2.2.1. 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 (bound to **C-@**) [*Command*]

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 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, the mark stack is popped without affecting the point.

The reason for these particular values for the prefix argument is that they can be generated easily 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 idiom will still work even if the default value is changed.

2.2.2. Using The Mouse

It can be convenient to use the mouse to point to positions in text, especially when moving large distances. Hemlock defines several commands for using the mouse. These commands can only be used when running under X windows (see page 6.)

Here to Top of Window (bound to **Rightdown**) [*Command*]
Top Line to Here (bound to **Leftdown**) [*Command*]

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 from the top of the window to the mouse cursor. Top Line to Here is the inverse operation, it scrolls backward, moving current the top line underneath the mouse.

If the mouse is near the left edge of a window, then these commands do smooth scrolling. Here To Top of Window repeatedly scrolls the window up by one line until the mouse button is released. Similarly, Top Line to Here smoothly scrolls down.

Point to Here (bound to **Middledown, S-Leftdown**) [*Command*]
Push Mark/Point to Here (bound to **Middleup, S-Leftup**) [*Command*]

Point to Here moves the point to the position of the mouse, changing to a different window if necessary.

When used in the window modeline, Point to Here moves the point of the window's buffer to the position within the file which corresponds percentagewise to the horizontal position of the mouse within the modeline.

Push Mark/Point to Here is a mouse command used to set the mark, and thus define the region. Note that this command is bound to **Middleup**, the releasing of 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 Push Mark/Point to Here 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.

Insert Kill Buffer (bound to **S-Rightdown**) [*Command*]

This command is a combination of Point to Here and Un-Kill (page 19). It moves the point to the mouse location and inserts the most recently killed text.

2.3. Modification Commands

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

2.3.1. 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 (bound to printing characters) [*Command*]
Self Insert inserts into the buffer the character which was typed to invoke it. 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 (bound to **Return**) [*Command*]
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 (bound to **C-q**) [*Command*]
Many characters cannot be inserted by Self Insert because they are bound to another command, or are otherwise magical (**C-g, Home**). **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 (bound to **C-o**) [*Command*]
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.

2.3.2. Deleting Characters

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

Delete Next Character (bound to **C-d**) [*Command*]
Delete Previous Character (bound to **Delete, Backspace**) [*Command*]
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.

2.3.3. 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.

2.3.4. Kill Ring Manipulation

Un-Kill (bound to **C-y**) [*Command*]
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 (bound to **M-y**) [*Command*]
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 (bound to **C-w**) [*Command*]
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 (bound to **M-w**) [*Command*]
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.

2.3.5. 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 (bound to **C-k**) [*Command*]
Backward Kill Line [*Command*]
Kill Line 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.

Backward Kill Line is similar, except that it kills from the point to the beginning of the line. If it is called at the beginning of the line, it kills the newline and any trailing whitespace on the previous line. With a prefix argument, this command is the same as Kill Line with a negated argument.

Kill Next Word (bound to **M-d**) [*Command*]
Kill Previous Word (bound to **M-Backspace, M-Delete**) [*Command*]
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.

2.3.6. Case Modification Commands

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

Capitalize Word (bound to **M-c**) [*Command*]
Lowercase Word (bound to **M-l**) [*Command*]
Uppercase Word (bound to **M-u**) [*Command*]

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.

Lowercase Region (bound to **C-x C-l**) [*Command*]
Uppercase Region (bound to **C-x C-u**) [*Command*]

These commands case-fold the text in the region. Since these commands can damage large amounts of text, they ask for confirmation before modifying large regions and can be undone with Undo.

2.3.7. 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 your friends and demonstrate your immense knowledge of exotic Emacs commands.

To the uninitiated, the behavior of transposition commands may seem mysterious; this has led some implementors to attempt to improve the definition of transposition, but right-thinking people will accept no substitutes. The Emacs transposition 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 (bound to **C-t**) [*Command*]

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 (bound to **C-x C-t**) [*Command*]

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 (bound to **M-t**) [*Command*]

This command transposes the previous word and the current or next word.

2.3.8. Whitespace Manipulation

These commands change the amount of space between words. See also the indentation commands in section 7.2.

Just One Space (bound to **M-\ **) [*Command*]
This command deletes all whitespace 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 (bound to **M-**) [*Command*]
This command deletes all blank characters around the point.

2.4. 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)
                              (remove #\; s)
                              s)))
```

2.5. 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 (initial value **:string-insensitive**) [Hemlock *Variable*]
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 (bound to **C-s**) [*Command*]
Reverse Incremental Search (bound to **C-r**) [*Command*]
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 it 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, then **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. Aborting the search inhibits the saving of the current search pattern as the last search string.

Escape 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. For example, typing **C-a** will exit the search *and* go to the beginning of the line.

Forward Search (bound to **M-s**) *[Command]*
Reverse Search (bound to **M-r**) *[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 (bound to **M-%**) *[Command]*

This command prompts in the echo area for a target string and a replacement string, and 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 page 12) at the current location. The search will be continued from wherever the point is left when the recursive edit is exited. This is useful for handling more complicated cases where a simple replacement will not achieve the desired effect.

Escape Exit without doing any replacement.

Home, C-_, ?, h
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`".

This command may be undone with Undo, but its undoing may not be undone.

Case Replace (initial value `t`) [Hemlock *Variable*]
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*]
This command is the same as Query Replace except that it doesn't ask any questions before doing replacements. After prompting for a target and replacement string, it replaces all occurrences of the target string following the point with the replacement string. If a prefix argument is specified, then only that many occurrences are replaced.

List Matching Lines [*Command*]
This command prompts for a search string and displays in a pop-up window all the lines containing the string that are after the point. If a prefix argument is specified, then that many lines before and after each matching line are also displayed.

Delete Matching Lines [*Command*]
Delete Non-Matching Lines [*Command*]
Delete Matching Lines prompts for a search string and deletes all lines containing the string that are after the point. Similarly, Delete Non-Matching Lines deletes all lines following the point that do not contain the specified string.

2.6. Page Commands

Another unit of text recognized by Hemlock is the page. A *page* is a piece of text delimited by formfeeds (`^L`.) The first non-blank line after the page marker is the *page title*. The page commands are be quite useful when logically distinct parts of a file are put on separate pages. See also Count Lines Page (page 24).

Previous Page (bound to **C-x [**) [*Command*]
Next Page (bound to **C-x]**) [*Command*]
Previous Page moves the point to the previous page delimiter, while Next Page moves to the next one. Any page delimiters next to the point are skipped. The prefix argument is a repeat count.

Mark Page (bound to **C-x C-p**) [*Command*]
This command puts the point at the beginning of the current page and the mark at the end. If given a prefix argument, marks the page that many pages from the current one.

Goto Page [*Command*]
This command does various different things, depending on the prefix argument:

no argument goes to the next page.

positive argument goes to an absolute page number, moving that many pages from the beginning of the

file.

zero argument prompts for string and goes to the page with that string in its title.

negative argument moves backward by that many pages, if possible.

View Page Directory [*Command*]

Insert Page Directory [*Command*]

View Page Directory uses a pop-up window to display the number and title of each page in the current buffer. Insert Page Directory is the same except that it inserts the text at the beginning of the buffer.

2.7. Counting Commands

Count Lines Region [*Command*]

This command displays the number of lines between the point and mark. The count includes the starting and ending lines.

Count Lines Page (bound to **C-x l**) [*Command*]

This command displays the number of lines in the current page and the number of lines before and after the point within that page. If given a prefix argument, the entire buffer is counted instead of just the current page.

Count Occurrences [*Command*]

This command prompts for a search string and displays the number of occurrences of the string that are after the point.

Chapter 3

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 to some text, a buffer has several other user-visible attributes:

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 page 4.

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 (bound to **C-x b**) [*Command*]

This command prompts for the name of a existing buffer and 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 (bound to **C-M-l**) [*Command*]
Circulate Buffers (bound to **C-M-L**) [*Command*]

With no prefix argument, Select Previous Buffer selects the buffer that has been selected most recently, similar to **C-x b Return**. If given a prefix argument, then it does the same thing as Circulate Buffers.

Circulate Buffers moves back into successively earlier buffers in the buffer history. If the previous command was not Circulate Buffers or Select Previous Buffer, then it does the same thing as Select Previous Buffer, otherwise it moves to the next most recent buffer. The original buffer at the start of the excursion is made the previous buffer, so Select Previous Buffer will always take you back to where you started.

These commands are generally used together. Often Select Previous Buffer will take you where you want to go. If you don't end up there, then using Circulate Buffers will do the trick.

Create Buffer (bound to **C-x M-b**) [*Command*]

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

Kill Buffer (bound to **C-x k**) [*Command*]

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 has been modified. This command is poorly named, since it has nothing to do with killing text.

List Buffers (bound to **C-x C-b**) [*Command*]

This command displays a list of all existing buffers in a pop-up window. A "*" is displayed before the name of each modified buffer, and the associated filename is displayed after the buffer name. If a buffer has no associated file then the number of lines in the buffer is displayed instead. When given a prefix argument, this command list only the modified buffers.

Buffer Not Modified (bound to **M-~**) [*Command*]

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 (bound to **C-x ~**) [*Command*]

This command displays a message indicating whether the current buffer is modified.

Set Buffer Read-Only [*Command*]

This command changes the flag that allows the current buffer to be modified. If a buffer is read-only, any attempt to modify it will result in an error. The buffer may be made writable again by repeating this command.

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 (bound to **C-x C-f**) [*Command*]

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*". This means that the filename "`/sys/emacs/teco.mid`" has "`Teco Mid /Sys/Emacs/`" as its the corresponding buffer name. 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 takes special action if the file has been modified on disk since it was read into Hemlock. This usually happens when several people are simultaneously editing a file, an unhealthy circumstance. If the buffer is unmodified, Find File just asks for confirmation before reading in the new version. If the buffer is modified, then Find File beeps and prompts for a single character to indicate what action to take. These characters are recognized:

Return, Space, y
: Prompt for a file to write the current buffer out to and then read in the new file.

Delete, Backspace, n
: Don't read in the new file.

r
: Read the new file into the old buffer, destroying any changes.

Save File (bound to **C-x C-s**) [*Command*]

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, which is made the associated file. If the buffer is not modified, then the user is asked whether to actually write it or not.

If the file has been modified on disk since the last time it was read, Save File prompts for confirmation before overwriting the file.

Save All Files (bound to **C-x C-m**) [*Command*]
Save All Files and Exit (bound to **C-x M-z**) [*Command*]
Save All Files Confirm (initial value `t`) [Hemlock *Variable*]

Save All Files does a Save File on all buffers which have an associated file. Save All Files and Exit does the same thing and then exits Hemlock.

When Save All Files Confirm is true, these commands will ask for confirmation before saving a file.

Visit File (bound to **C-x C-v**) [*Command*]

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 (bound to **C-x C-w**) [*Command*]

This command prompts for a file and writes the current buffer out to it, changing the associated filename and resetting then modification flag. When the buffer's associated file is specified this command does the same thing as Save File.

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.

Since Backup File doesn't update the write date for the buffer, Find File and Save File will get all upset if you back up a buffer on any file has been read into Hemlock.

Revert File [*Command*]

Revert File Confirm (initial value `t`) [Hemlock *Variable*]

This command replaces the text in the current buffer with the contents of the associated file and clears the modified flag. The point is put in approximately the same place that it was before the file was read.

If the buffer is modified and Revert File Confirm is true, then the user is asked for confirmation.

Insert File (bound to **C-x C-r**) [*Command*]

This command prompts for a file and inserts it at the point.

Write Region [*Command*]

This command prompts for a file and writes the text in the region out to it.

Add Newline at EOF on Writing File (initial value `:ask-user`) [Hemlock *Variable*]

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 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 of the default type. This permits the creation of a file having no type, in this case "`foo`".

Pathname Defaults (initial value `(pathname "gazonk.del")`) [Hemlock *Variable*]
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 "*option*: *value*" pairs separated by semicolons. A typical example:

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

These options are currently defined:

Mode	The argument is the name of the major mode to put the buffer in when the file is read.
Package	The argument is the name of the package to be used for reading code in the file. This is only meaningful for LISP code (see page 56.)
Editor	If the argument is true, this option turns on the Editor minor mode in the current buffer (see page 59.)
Log	The argument is the name of the change log file associated with this file (see page 39.)

If the option list contains no ":" then the entire string is used as the name of the major 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. The buffer must have an associated file.

3.3.3. File Utility Commands

These commands allow the filesystem to be manipulated from within the editor.

Delete File [*Command*]
Rename File [*Command*]
Delete File prompts for the name of a file, and deletes it. Rename File prompts for the name of a file and a name to rename it to.

Directory [*Command*]
This command does a directory into a pop-up window. It prompts for a pathname, which may contain wildcards in the name and type.

3.4. Windows

When running under X windows, each Hemlock window is a separate window, so window manager commands can be used to move and reshape windows.

New Window (bound to **C-x C-n**) [*Command*]
Split Window (bound to **C-x 2**) [*Command*]
Stack Window [*Command*]

New Window creates a new window on the screen that displays the current buffer. The dimensions of the window are determined by asking the user.

Split Window is the same as New Window except that the new window is created by splitting the current window in half. If the current window is too small to be reasonably split, something else is done.

Stack Window is the same as *New Window*, except that it makes a window exactly superimposed on the current window.

Next Window (bound to **C-x n**) [*Command*]
Previous Window (bound to **C-x p**) [*Command*]

These commands make the next or previous window the new current window, often changing the current buffer in the process. When a window is created, it is arbitrarily made the next window of the current window. The location of the next window is, in general, unrelated to that of the current window.

Delete Window (bound to **C-x C-d, C-x d**) [*Command*]
Delete Next Window (bound to **C-x 1**) [*Command*]

Delete Window makes the current window go away, making the next window current. Delete Next Window deletes the next window, leaving the current window unaffected.

Line to Top of Window (bound to **M-!**) [*Command*]
Line to Center of Window (bound to **M-#**) [*Command*]

Line to Top of Window scrolls the current window up until the current line is at the top of the screen.

Line to Center of Window attempts to scroll the current window so that the current line is vertically centered.

Scroll Next Window Down (bound to **C-M-v**) [*Command*]
Scroll Next Window Up (bound to **C-M-V**) [*Command*]

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

Refresh Screen (bound to **C-l**) [*Command*]

This command refreshes all windows, which is useful if the screen got trashed. The current line is centered within the current window.

Chapter 4

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.

Text Mode [*Command*]
: This commands puts the current buffer into "Text" mode.

4.1. Sentence Commands

A sentence is defined as a sequence of characters ending with a period, question mark or exclamation point, followed by either two spaces or a newline. A sentence may also be terminated 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 (bound to **M-a**) [*Command*]
Backward Sentence (bound to **M-e**) [*Command*]
: Forward Sentence moves the point forward past the next sentence end. Backward Sentence moves to the beginning of the current sentence. A prefix argument may be used as a repeat count.

Forward Kill Sentence (bound to **M-k**) [*Command*]
Backward Kill Sentence (bound to **C-x Delete, C-x Backspace**) [*Command*]
: 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 "‘" or ".", in which case the delimiting line is not part of the paragraph. Other characters may be paragraph delimiters in some modes. 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 (bound to **M-]**) [*Command*]
Backward Paragraph (bound to **M-[**) [*Command*]
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 (bound to **M-h**) [*Command*]
This command puts the point at the beginning and the mark at the end of the current paragraph.

Paragraph Delimiter Function (initial value `default-para-delim-function`) [Hemlock *Variable*]
This variable holds a function that takes a mark as its argument and returns true when the line it points to should break the paragraph.

4.3. Filling

Filling is a coarse text-formatting process which attempts to make all the lines roughly the same length, but doesn't vary 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 (bound to **C-x f**) [*Command*]
This command sets the fill column to the column that the point is currently at, or the one specified by the absolute value of prefix argument, if it is supplied. The fill column is the column past which no text is permitted to extend.

Set Fill Prefix (bound to **C-x .**) [*Command*]
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 or comments.

Fill Column (initial value `75`) [Hemlock *Variable*]
Fill Prefix (initial value `nil`) [Hemlock *Variable*]
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 (bound to **M-q**) [*Command*]
This command fills the text in the current or next paragraph. The point is not moved.

Fill Region (bound to **M-g**) [*Command*]
This command fills the text in the region. Since filling can mangle a large quantity of text, this command asks for confirmation before filling a large region (see Region Query Size.)

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 (bound to **Linefeed** in Fill mode) [*Command*]
Auto Fill Return (bound to **Return** in Fill mode) [*Command*]

Auto Fill Linefeed fills the current line if it needs it and then goes to a new line and inserts the fill prefix. Auto Fill Return is similar, but does not insert the fill prefix on the new line.

Auto Fill Space (bound to **Space** in Fill mode) [*Command*]

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.

Auto Fill Space Indent (initial value `nil`) [Hemlock *Variable*]

This variable determines how lines are broken by the auto fill commands. If it is true, new lines are created using the Indent New Comment Line command, otherwise the New Line command is used. Language modes should define this variable to be true so that auto fill mode can be used on code.

4.4. Scribe Mode

Scribe mode provides a number of facilities useful for editing Scribe documents. It is also sufficiently parameterizable to be adapted to other similar syntaxes.

Scribe Mode [*Command*]

This command puts the current buffer in Scribe mode. Except for special Scribe commands, the only difference between Scribe mode and Text mode is that the rules for determining paragraph breaks are different. In Scribe mode, paragraphs delimited by Scribe commands normally placed on their own line, in addition to the normal paragraph breaks. The main reason for doing this is that it prevents Fill Paragraph from mashing these commands into the body of a paragraph.

Insert Scribe Directive (**C-h** in Scribe mode) [*Command*]

This command prompts for a single character sub-command to determine which Scribe directive to insert. Directives are inserted differently depending on their kind:

environment — The entire current paragraph is enclosed in a begin-end pair: `@begin[`*directive*`]` *paragraph* `@end[`*directive*`]`.

command — The previous word is enclosed by `@`*directive*`[`*word*`]`. If the previous word is already enclosed by a use of the same command, then the beginning of the command is extended backward by one word.

Typing **Home** or **C-_** to this command's prompt will display a list of all the defined command characters.

Add Scribe Directive [*Command*]

This command adds to the database of directives recognized by the Insert Scribe Directive command. It prompts for the directive's name, the kind of directive (environment or command) and the sub-command character to use.

Add Scribe Paragraph Delimiter [*Command*]

This command prompts for a string to add to the list of commands that delimit paragraphs in Scribe mode. If a prefix argument is supplied, then the command is made to no longer be a delimiter.

Escape Character (initial value #\@) [Hemlock *Variable*]
Close Paren Character (initial value #\]) [Hemlock *Variable*]
Open Paren Character (initial value #\[) [Hemlock *Variable*]
These variables determine the characters used when a Scribe directive is inserted.

Scribe Bracket Table [Hemlock *Variable*]
This variable holds a **simple-vector** indexed by character codes. If a character is a bracket, then the entry for its **char-code** should be the opposite bracket. If a character is not a bracket, then the entry should be **nil**.

4.5. Spelling Correction

Hemlock has a spelling correction facility based on the dictionary for the ITS spell program. This dictionary is fairly small, having only 45,000 word 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 (bound to **M-$**) [*Command*]
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.

Spell Ignore Uppercase (initial value `nil`) [Hemlock *Variable*]

If this variable is true, then Check Word Spelling and Correct Buffer Spelling will ignore unknown words that are all uppercase. This is useful for acronyms and cryptic formatter directives.

Add Word to Spelling Dictionary (bound to **C-x $**) [*Command*]

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. Except for Augment Spelling Dictionary, all the commands that add words to the dictionary put their insertions in this list.

4.5.1. 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 (bound to word delimiters in Spell mode) [*Command*]
Check Word Spelling Beep (initial value `t`) [Hemlock *Variable*]
Correct Unique Spelling Immediately (initial value `t`) [Hemlock *Variable*]

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.

If Check Word Spelling Beep is true, then this command will beep when an unknown word is found. If Correct Unique Spelling Immediately is true, then this command will immediately attempt to correct any unknown word, automatically making the correction if there is only one possible.

Undo Last Spelling Correction (bound to **C-x a**) [*Command*]

This command undoes the last incremental spelling correction. The "correction" is replaced with the old word, and the old word is inserted in the dictionary. Any automatic replacement for the old word is eliminated.

Correct Last Misspelled Word [*Command*]

This command places the cursor 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.
Backspace, Delete, n	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.
Escape	Exit and forget about this word.

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

Chapter 5

Managing Large Systems

Hemlock provides three 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.
3. A change log facility, which maintains a single file containing a record of the edits done on a system.

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 in the following format:

- 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.

This syntax is used for historical reasons. 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. The Compile Group command is described on page 57.

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:

Escape, Space, y
Exit Group Search.

Delete, Backspace, n
Continue searching for the next occurrence of the string.

! Continue the search at the beginning of the next file, skipping the remainder of the current file.

C-r Go into a recursive edit at the current location, and continue the search when it is exited.

Group Find File (initial value `nil`) [Hemlock *Variable*]

If this variable is true, the group searching and replacing commands will read each file into its own buffer using Find File. Since this may result in large amounts of memory being consumed by unwanted files, this variable is false by default. When this variable is false, each file which must be read is initially put in the Group Search buffer. If the file is modified and the changes are saved, then the file is given its own buffer.

Group Save File Confirm (initial value `t`) [Hemlock *Variable*]

If this variable is true, the group replacing commands will ask for confirmation before saving any modified file.

5.2. Source Comparison

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.

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. There are various variables that control exactly how the comparison is done.

If a prefix argument is specified, then only only the lines in the the regions of the two buffers are compared.

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 the string "`**** MERGE LOSSAGE ****`" followed by both versions. 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 (initial value `nil`) [Hemlock *Variable*]
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 (initial value `t`) [Hemlock *Variable*]
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 (initial value `3`) [Hemlock *Variable*]
This variable controls the number of lines Compare Buffers and Merge Buffers will compare when resynchronizing after a difference has been encountered.

5.3. Change Logs

The Hemlock change log facility encourages the recording of changes to a system by making it easy to do so. The change log is kept in a separate file so that it doesn't clutter up the source code. The name of the log for a file is specified by the `Log` file option (see page 29.)

Log Change [*Command*]
Log Entry Template [Hemlock *Variable*]
Log Change makes a new entry in the change log associated with the file. Any changes in the current buffer are saved, and the associated log file is read into its own buffer. The name of the log file is determined by merging the name specified in the `Log` option with the current buffer's file name, so it is not usually necessary to put the full name there. After inserting a template for the log entry at the beginning of the buffer, the command enters a recursive edit (see page 12) so that the text of the entry may be filled in. When the user exits the recursive edit, the log file is saved.

The variable "Log Entry Template" determines the format of the change log entry. Its value is a COMMON LISP `format` control string. The format string is passed three string arguments: the full name of the file, the creation date for the file and the name of the file author. If the creation date is not available, the current date is used. If the author is not available then `nil` is passed. If there is an `@` in the template, then it is deleted and the point is left at that position.

Chapter 6

Special Modes

6.1. Overwrite Mode

Overwrite mode is a minor mode which is useful for creating figures and tables out of text. In this mode, typing a printing replaces the character at the point instead of inserting the character. **C-q** can be used to insert characters normally.

Overwrite Mode [*Command*]
: This command turns on Overwrite mode in the current buffer. If it is already on, then it is turned off. A positive argument turns Overwrite mode on, while zero or a negative argument turns it off.

Self Overwrite (bound to printing characters in Overwrite mode) [*Command*]
: This command replaces the next character with the character used to invoke it and then moves forward. If the next character is a tab, it is expanded into the appropriate number of spaces. At the end of the line, it inserts the character.

Overwrite Delete Previous Character (bound to **Delete** and **Backspace** in Overwrite mode) [*Command*]
: This command replaces the previous character with a space and moves backwards. Tabs and newlines are deleted.

6.2. Word Abbreviation

Word abbreviation provides a way to speed the typing of frequently used words and phrases. When in Abbrev mode, typing a word delimiter causes the previous word to be replaced with its *expansion* if there is one currently defined. The expansion for an abbrev may be any string, so this mode can be used for abbreviating programming language constructs and other more obscure uses. For example, Abbrev mode can be used to automatically correct common spelling mistakes and to enforce consistent capitalization of identifiers in programs.

Abbrev is an abbreviation for *abbreviation*, which is used for historical reasons. Obviously the original writer of Abbrev mode hated to type long words and could hardly use Abbrev mode while writing Abbrev mode.

A word abbrev can be either global or local to a major mode. A global word abbrev is defined no matter what the current major mode is, while a mode word abbrev is only defined when its mode is the major mode in the current buffer. Mode word abbrevs can be used to prevent abbrev expansion in inappropriate contexts.

6.2.1. Basic Commands

Abbrev Mode [*Command*]

This command turns on Abbrev mode in the current buffer. If Abbrev mode is already on, it is turned off. Abbrev mode must be on for the automatic expansion of word abbrevs to occur, but the abbreviation commands are bound globally and may be used at any time.

Abbrev Expand Only (bound to word-delimiters in Abbrev mode) [*Command*]

This is the word abbrev expansion command. If the word before the point is a defined word abbrev, then it is replaced with its expansion. The replacement is done using the same case-preserving heuristic as is used by Query Replace. This command is globally bound to **M-Space** so that abbrevs can be expanded when Abbrev mode is off. An undesirable expansion may be inhibited by using **C-q** to insert the delimiter.

Inverse Add Global Word Abbrev (bound to **C-x** -) [*Command*]
Inverse Add Mode Word Abbrev (bound to **C-x C-h, C-x Backspace**) [*Command*]

Inverse Add Global Word Abbrev prompts for a string and makes it the global word abbrev expansion for the word before the point.

Inverse Add Mode Word Abbrev is identical to Inverse Add Global Word Abbrev except that it defines an expansion which is local to the current major mode.

Make Word Abbrev [*Command*]

This command defines an arbitrary word abbreviation. It prompts for the mode, abbreviation and expansion. If the mode "`Global`" is specified, then it makes a global abbrev.

Add Global Word Abbrev (bound to **C-x +**) [*Command*]
Add Mode Word Abbrev (bound to **C-x C-a**) [*Command*]

Add Global Word Abbrev prompts for a word and defines it to be a global word abbreviation. The prefix argument determines which text is used as the expansion:

no prefix argument The word before the point is used as the expansion of the abbreviation.

zero prefix argument
The text in the region is used as the expansion of the abbreviation.

positive prefix argument
That many words before the point are made the expansion of the abbreviation.

negative prefix argument
Do the same thing as Delete Global Word Abbrev instead of defining an abbreviation.

Add Mode Word Abbrev is identical to Add Global Word Abbrev except that it defines or deletes mode word abbrevs in the current major mode.

Word Abbrev Prefix Mark (bound to **M-"**) [*Command*]

This command allows Abbrev Expand Only to recognize abbreviations when they have prefixes attached. First type the prefix, then use this command. A hyphen (-) will be inserted in the buffer. Now type the abbreviation and the word delimiter. Abbrev Expand Only will expand the abbreviation and remove the hyphen.

Note that there is no need for a suffixing command, since Abbrev Expand Only may be used explicitly by typing **M-Space**.

Unexpand Last Word (bound to **C-x u**) [*Command*]
This command undoes the last word abbrev expansion. If repeated, undoes its own effect.

6.2.2. Word Abbrev Files

A word abbrev file is a file which holds word abbrev definitions. Word abbrev files allow abbrevs to be saved so that they may be used across many editing sessions.

Read Word Abbrev File [*Command*]
This command reads in a word abbrev file, adding all the definitions to those currently defined. If a definition in the file is different from the current one, the current definition is replaced.

Write Word Abbrev File [*Command*]
This command prompts for a file and writes all currently defined word abbrevs out to it.

Append to Word Abbrev File [*Command*]
This command prompts for a word abbrev file and appends any new definitions to it. An abbrev is new if it has been defined or redefined since the last use of this command. Definitions made by reading word abbrev files are not considered.

6.2.3. Listing Word Abbrevs

List Word Abbrevs [*Command*]
Word Abbrev Apropos [*Command*]
List Word Abbrevs displays a list of each defined word abbrev, with its mode and expansion.

Word Abbrev Apropos is similar, except that it only displays abbrevs which contain a specified string, either in the definition, expansion or mode.

6.2.4. Editing Word Abbrevs

Word abbrev definition lists are edited by editing the text representation of the definitions. Word abbrev files may be edited directly, like any other text file. The set of abbrevs currently defined in Hemlock may be edited using the commands described in this section.

The text representation of a word abbrev is fairly simple. Each definition begins at the beginning of a line. Each line has three fields which are separated by ASCII tab characters. The fields are the abbreviation, the mode of the abbreviation and the expansion. The mode is represented as the mode name inside of parentheses. If the abbrev is global, then the mode field is empty. The expansion is represented as a quoted string since it may contain any character. The string is quoted with double-quotes ("); double-quotes in the expansion are represented by doubled double-quotes. The expansion may contain newline characters, in which case the definition will take up more than one line.

Edit Word Abbrevs [*Command*]
This command inserts the current word abbrev definitions into the Edit Word Abbrevs buffer and then enters a recursive edit on the buffer. When the recursive edit is exited, the definitions in the buffer become the new current abbrev definitions.

Insert Word Abbrevs [*Command*]

This command inserts at the point the text representation of the currently defined word abbrevs.

Define Word Abbrevs [*Command*]

This command interprets the text of the current buffer as a word abbrev definition list, adding all the definitions to those currently defined.

6.2.5. Deleting Word Abbrevs

Word abbrevs may be deleted either individually or collectively. Individual abbrev deletion neutralizes single abbrevs which have outlived their usefulness; collective deletion provides a clean slate from which to initiate abbreviatory activities.

Delete All Word Abbrevs [*Command*]

This command deletes all word abbrevs which are currently defined.

Delete Global Word Abbrev [*Command*]
Delete Mode Word Abbrev [*Command*]

Delete Global Word Abbrev prompts for a word abbreviation and deletes its global definition. If given a prefix argument, deletes all global abbrev definitions.

Delete Mode Word Abbrev is identical to Delete Global Word Abbrev except that it deletes definitions in the current major mode.

Chapter 7

Editing Programs

7.1. Comment Manipulation

Hemlock has commenting commands which can be used in almost any language. The behavior of these commands is determined by several Hemlock variables which language modes should define appropriately.

Indent for Comment (bound to **M-;**) [*Command*]

This is the most basic commenting command. If there is already a comment on the current line, then the point is moved to the start of the comment. If there no comment, an empty one is created. Normally the comment is indented so that it starts at the Comment Column.

The comment is not indented to the comment column in these cases:

1. If the comment currently starts at the beginning of the line or if the last character in the Comment Start appears three times, then the comment is not moved.
2. If the last character in the Comment Start appears two times, then the comment is indented like a line of code.
3. If the text on the line prevents the comment from being placed in the desired position, it is placed at the end of the line, separated from the text by a space.

Although the rules about replication in the comment start are oriented toward LISP commenting styles, they can be exploited in other languages.

When given a prefix argument, this command indents any existing comment on that many consecutive lines. This is useful for fixing up the indentation of a group of comments.

Indent New Comment Line (bound to **M-j, M-Linefeed**) [*Command*]

This commend ends the current comment and starts a new comment on a blank line, indenting the comment the same way that Indent for Comment does. When not in a comment, this command is the same as Indent New Line.

Up Comment Line (bound to **M-p**) [*Command*]
Down Comment Line (bound to **M-n**) [*Command*]

These commands are similar to Previous Line or Next Line followed by Indent for Comment. Any empty comment on the current line is deleted before moving to the new line.

Kill Comment (bound to **C-M-;**) [*Command*]

This command kills any comment on the current line. When given a prefix argument, it kills comments on that many consecutive lines. Undo will restore the unmodified text.

Set Comment Column (bound to **C-x ;**) [*Command*]
This command sets the comment column to its prefix argument. If used without a prefix argument, it sets the comment column to the column the point is at.

Comment Start (initial value `nil`) [Hemlock *Variable*]
Comment End (initial value `nil`) [Hemlock *Variable*]
Comment Begin (initial value `nil`) [Hemlock *Variable*]
Comment Column (initial value `0`) [Hemlock *Variable*]
These variables determine the behavior of the comment commands.

Comment Start — The string which indicates the start of a comment. If this is `nil`, then there is no defined comment syntax.

Comment End — The string which ends a comment. If this is `nil`, then the comment is terminated by the end of the line.

Comment Begin — The string inserted to begin a new comment.

Comment Column — The column that normal comments start at.

7.2. Indentation

Nearly all programming languages have conventions for indentation: whitespace at the beginning of lines. The Hemlock indentation facility is integrated into the command set so that in interacts well with other features such as filling and commenting.

Indent (bound to **Tab, C-i**) [*Command*]
This command indents the current line. With a prefix argument, indents that many lines and moves down. Exactly what constitutes indentation depends on the current mode (see Indent Function).

Quote Tab (bound to **M-Tab**) [*Command*]
This command inserts a tab character.

Indent New Line (bound to **Linefeed**) [*Command*]
This command starts a new indented line. Deletes any whitespace before the point and inserts indentation on a blank line. The effect of this is similar to **Return** followed by **Tab**. The prefix argument is passed to New Line, which is used to insert the blank line.

Back to Indentation (bound to **M-m, C-M-m**) [*Command*]
This command moves point to the first non-whitespace character on the current line.

Delete Indentation (bound to **M-^, C-M-^**) [*Command*]
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. Usually one space is left between the two joined line fragments, but there are several exceptions.

The non-whitespace immediately surrounding the deleted line break determine the amount of space inserted.

1. If the preceding character is an "`(`" or the following character is a "`)`", then no space is inserted.
2. If the preceding character is a newline, then no space is inserted. This will happen if the

previous line was blank.

3. If the preceding character is a sentence terminator, then two spaces are inserted.

When given a prefix argument, this command joins the current and next lines, rather than the previous and current lines.

Indent Rigidly (bound to **C-x Tab, C-x C-i**) [*Command*]
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.

Indent Region (bound to **C-M-**) [*Command*]
This command indents every line in the region. It may be undone with Undo.

Indent Function (initial value `tab-to-tab-stop`) [Hemlock *Variable*]
The value of this variable determines how indentation is done. The value is a function which is passed a mark as its argument. The function should indent the line which the mark points to. The function may move the mark around on the line. The mark will be `:left-inserting`.

Indent with Tabs (initial value `indent-using-tabs`) [Hemlock *Variable*]
Spaces per Tab (initial value 8) [Hemlock *Variable*]
Indent with Tabs holds a function that takes a mark and a number of spaces and inserts tabs and spaces to indent that number of spaces. The default definition uses Spaces per Tab to determine the size of a tab.

7.3. Language Modes

Hemlock's language modes are currently fairly crude, but probably provide better programming support than most non-extensible editors.

Pascal Mode [*Command*]
This command sets the current buffer's major mode to Pascal. Pascal mode borrows parenthesis matching from Scribe mode and indents lines under the previous line.

Chapter 8

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.

8.1. Lisp Mode

Lisp mode is a major mode used for editing LISP code. Although most LISP specific commands are globally bound, Lisp mode is necessary to enable LISP indentation, commenting and parenthesis-matching.

Lisp Mode [*Command*]
: Set the major mode of the current buffer to Lisp.

8.2. 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 (bound to **C-M-f**) [*Command*]
Backward Form (bound to **C-M-b**) [*Command*]
: 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 (bound to **C-M-k**) [*Command*]
Backward Kill Form (bound to **C-M-Delete, C-M-Backspace**) [*Command*]
: 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 (bound to **C-M-@**) [*Command*]
: This command sets the mark at the end of the current or next form.

Transpose Forms (bound to **C-M-t**) [*Command*]
: 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 () (bound to **M-(**) *[Command]*

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.

8.3. List Manipulation

List commands are similar to form commands, but they 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 (bound to **C-M-n**) *[Command]*

Backward List (bound to **C-M-p**) *[Command]*

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 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 (bound to **C-M-)**) *[Command]*

Backward Up List (bound to **C-M-(, C-M-u**) *[Command]*

Forward Up List moves to 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 (bound to **C-M-d**) *[Command]*

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

Extract List (bound to **C-M-x**) *[Command]*

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.

8.4. 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 other top level forms such as a `defstruct` and `defmacro` work just as well.

End of Defun (bound to **C-M-e, C-M-]**) *[Command]*

Beginning of Defun (bound to **C-M-a, C-M-[**) *[Command]*

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 (bound to **C-M-h**) [*Command*]
This command puts the point at the beginning and the mark at the end of the current or next defun.

8.5. 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. See section 7.2 for a description of the general-purpose indentation commands. These are the indentation rules used:

- If in a semicolon (`;`) comment, then the standard comment indentation rules are used. See page 45.
- If in a quoted string, copy the indentation from the previous line.
- If there is no enclosing list, then use no indentation.
- If enclosing list resembles a call to a known macro or special-form, then the first few arguments are given greater indentation and the first body form is indented two spaces. If the first special argument is on the same line as the beginning of the form, then following special arguments will be indented to the start of the first special argument, otherwise all special arguments are indented four spaces.
- If the previous form starts on its own line, then the indentation is copied form that form. This rule allows the default indentation to be overridden: once a form has been manually indented to the user's satisfaction, subsequent forms will be indented in the same way.
- If the enclosing list has some arguments on the same line as the form start, then subsequent arguments will be indented to the start of the first argument.
- If the enclosing list has no argument on the same line as the form start, then arguments will be indented one space.

Indent Form (bound to **C-M-q**) [*Command*]
This command indents all the lines in the current form, leaving the point unmoved.

Defindent (bound to **C-M-#**) [*Command*]
This command prompts for the number of special arguments to associate with the symbol at the beginning of the current or containing list.

Indent Defanything (initial value 2) [Hemlock *Variable*]
This is the number of special arguments implicitly assumed to be supplied in calls to functions whose names begin with "`def`". If set to `nil`, this feature is disabled.

Move Over) (bound to **M-)**) [*Command*]
This command moves past the next close parenthesis and then does the equivalent of Indent New Line.

8.6. 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) (bound to **(** in Lisp mode)
Paren Pause Period [*Command*]
[Hemlock *Variable*]
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 beep. If the matching parenthesis is off the top of the screen, then the line on which it appears is displayed in the echo area.

8.7. Parsing Lisp

LISP mode has fairly complete knowledge of LISP syntax, but since it does not use the reader, and must work incrementally, it can be confused by legal constructs. LISP mode totally ignores the read-table, so user-defined read macros have no effect on the editor. In some cases, the values the Lisp Syntax character attribute can be changed to get a similar effect.

LISP commands consistently treat semicolon (`;`) style comments as whitespace when parsing, so a LISP command used in a comment will affect the next (or previous) form outside of the comment. Since `#|` ... `|#` comments are not recognized, they can used to comment out code, but still allowing LISP editing commands to be used.

Strings are parsed similarly to symbols. When within a string, the next form is after the end of the string, and the previous form is the beginning of the string.

Defun Parse Goal (initial value **2**) [Hemlock *Variable*]
Maximum Lines Parsed (initial value **250**) [Hemlock *Variable*]
Minimum Lines Parsed (initial value **25**) [Hemlock *Variable*]

In order to save time, LISP mode does not parse the entire buffer every time a LISP command is used. Instead, it uses a heuristic to guess the region of the buffer that is likely to be interesting. These variables control the heuristic.

Normally, parsing begins and ends on defun boundaries (an open parenthesis at the beginning of a line). Defun Parse Goal specifies the number of defuns before and after the point to parse. If this parses fewer lines than Minimum Lines Parsed, then parsing continues until this lower limit is reached. If we cannot find enough defuns within Maximum Lines Parsed lines then we stop on the farthest defun found, or at the point where we stopped if no defuns were found.

When the heuristic fails, and does not parse enough of the buffer, then command usually acts as though a syntax error was detected. If the parse starts in a bad place (such as in the middle of a string), then LISP commands will be totally confused. Such problems can usually be eliminated by increasing the values of some of these variables.

Parse Start Function (initial value **`start-of-parse-block`**) [Hemlock *Variable*]
Parse End Function (initial value **`end-of-parse-block`**) [Hemlock *Variable*]

These variables determine the region of the buffer parsed. The values are functions that take a mark and move it to the start or end of the parse region. The default values implement the heuristic described above.

Chapter 9

Interacting With Lisp

LISP encourages highly interactive programming environments by requiring decisions about object type and function definition to be postponed until run time. Hemlock supports interactive programming in LISP by providing incremental redefinition and environment examination commands. Hemlock also uses Mach IPC to support multiple LISP processes, each of which may be on any machine.

9.1. Eval Servers

Hemlock runs in the editor process and interacts with other LISP processes called *eval servers*. A user's LISP program normally runs in an eval server process. The separation between editor and eval server has several advantages:

- The editor is protected from any bad things which may happen while a LISP program is being debugged.
- Editing may be done while a LISP program is running.
- The eval server may be on a different machine, removing the load from the editing machine.
- Multiple eval servers allow several totally distinct LISP environments to be maintained.

Instead of providing an interface to a single LISP environment, Hemlock coordinates multiple LISP environments.

Eval servers are referred to by name so that there is a convenient way to discriminate between servers when the editor is connected to more than one.

Prompt for Slave Name (initial value `t`) [Hemlock *Variable*]

This variable controls how eval servers are named. When true, the user is prompted for the eval server name to give to the slave. When false, names of the form `Slave` *n* are automatically generated. *n* is a small integer obtained from a counter.

If the LISP process for an eval server terminates, then that eval server is said to be dead. A message is displayed in the echo area whenever an eval server dies. If a command attempts to use a dead eval server, then the command will beep and display a message. An eval server can be resurrected by creating a new LISP process with the same eval server name.

9.1.1. Slaves

An eval server may be either a *slave* or a *registered eval server*. Since most eval servers are slaves, registered eval servers are discussed separately in section 9.8. A slave is a LISP process that uses a typescript (see page 55) to run its top-level **`read eval print`** loop in a Hemlock buffer. The buffer that a slave uses for I/O is called its *slave buffer*. The name of the slave buffer is the same as the eval server's name.

Hemlock creates a *background buffer* for each eval server that it is connected to. The background buffer's name is Background *name*, where *name* is the name of the eval server. Slaves direct compiler warning output to the background buffer to avoid cluttering up the slave buffer.

9.1.2. The Current Eval Server

Although Hemlock can be connected to several eval servers simultaneously, one eval server is designated as the *current eval server*. This is the eval server used to handle evaluation and compilation requests. The current eval server is normally globally specified, but it may also be shadowed locally in specific buffers. In slave and background buffers, the current eval server is the eval server associated with that buffer.

Set Eval Server [*Command*]
Set Buffer Eval Server [*Command*]
Set Eval Server prompts for the name of an eval server and makes it the the current eval server. Set Buffer Eval Server is the same except that is sets the eval server for the current buffer only. See also Set Compile Server (page 57).

When Hemlock first starts up, there isn't any current eval server. The current eval server is also made undefined when the background buffer for the current eval server is deleted. If there isn't a current eval server, commands that need to use the current eval server will create a slave and make it the current eval server.

Confirm Slave Creation (initial value `t`) [Hemlock *Variable*]
If this variable is true, then confirmation will be requested before automatically creating a slave.

Select Slave (bound to **C-M-c**) [*Command*]
This command changes the current buffer to the current eval server's slave buffer. If the current eval server is not a slave, then beep. If there is no current eval server, then create a slave and make it the current eval server. If a prefix argument is supplied, then create a new slave regardless of whether there is a current eval server or not.

The slave buffer is a typescript (see page 55) that is used for the slave's top-level **`read eval print`** loop.

Select Background (bound to **C-M-C**) [*Command*]
This command changes the current buffer to the current eval server's background buffer. If there is no current eval server, then beep.

9.1.3. Eval Server Operations

Hemlock handles requests for compilation or evaluation by queuing an *operation* on the current eval server. Any number of operations may be queued, but each eval server can only service one operation at a time. Information about the progress of operations is displayed in the echo area.

Abort Operations (bound to **C-c a**) [*Command*]
This command aborts all operations on the current eval server, either queued or in progress. Any operations already in the **`Aborted`** state will be flushed.

List Operations (bound to **C-c l**) [*Command*]

This command lists all operations which have not yet completed. Along with a description of the operation, the state and eval server is displayed. The following states are used:

`Unsent`	The operation is in local queue in the editor, and hasn't been sent yet.
`Pending`	The operation has been sent, but has not yet started execution.
`Running`	The operation is currently being processed.
`Aborted`	The operation has been aborted, but the eval server has not yet indicated termination.

9.2. Typescripts

Both slave buffers and background buffers are typescripts. The typescript protocol allows other processes to do stream-oriented "terminal" interaction in a Hemlock buffer. When there is a typescript in a buffer, the Typescript minor mode will be present. Some of the commands described in this section are also used by Eval mode (page 59.)

Typescripts are simple to use. Output from the process is inserted into the buffer. To give the process input, use any combination of Hemlock commands to insert the input at the end of the buffer, and then type **Return** to cause the input to be sent to the process.

Confirm Typescript Input (bound to **Return** in Typescript mode) [*Command*]

This command sends text that has been inserted at the end of the current buffer to the process reading on the buffer's typescript. Before sending the text, the point is moved to the end of the buffer and a newline is inserted.

Input may be edited as much as is desired before it is confirmed; the result of editing input after it has been confirmed is unpredictable. For this reason, it is desirable to postpone confirming of input until it is actually complete. The Indent New Line command is often useful for inserting newlines without confirming the input.

If there isn't any pending request for input, then the text is queued instead of being sent immediately. Any number of inputs may be typed ahead in this fashion. Hemlock makes sure that the inputs and outputs get interleaved correctly so that when all input has been read, the buffer looks the same as it would have if the input had not been typed ahead.

Kill Interactive Input (bound to **M-i** in Typescript and Eval modes) [*Command*]

This command kills any input that would have been confirmed by **Return**.

Abort Typescript Input (bound to **C-M-i** in Typescript mode) [*Command*]

This command moves to the end of the buffer and discards any input which has not been read yet.

Next Interactive Input (bound to **M-n** in Typescript and Eval modes) [*Command*]
Previous Interactive Input (bound to **M-p** in Typescript and Eval modes) [*Command*]

A history of interactive inputs 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.

Interactive Beginning of Line (bound to **C-a** in Typescript and Eval modes) [*Command*]

This command is identical to Beginning of Line unless there is no prefix argument and the point is on the same line as the start of the current input; then it moves to the beginning of the input. This is useful since it skips over any prompt which may be present.

Input Wait Alarm (initial value `:loud-message`) [Hemlock *Variable*]

This variable determines what action is taken when a process goes into an input wait on a typescript that isn't currently displayed in any window. These values are legal:

`:loud-message` Beep and display a message in the echo area indicating which buffer is waiting for input.

`:message` Display a message, but don't beep.

`nil` Don't do anything.

Process Control [*Command*]

Some typescripts have additional information attached which allows Hemlock to control the process which uses the typescript. This command reads a single character option which determines the action taken:

b Put the current process in a break loop so that it can be debugged. This is similar in effect to an interrupt signal (^C or ^\ in Hemlock.)

g Cause the current process to throw to the top-level `read eval print` loop. This is similar in effect to a quit signal (^\).

p Call the `ext:abort` function in the current process. This is the same as **g** unless some program has established a `ext:catch-abort` handler, in which case it will abort back to that point.

a Calls the Abort Operations command.

l Calls the List Operations command.

9.3. The Current Package

The current package is the package which LISP interaction commands use. Unless a local package has been specified for the current buffer, the current package is determined by the value of `*package*` in the editor. If the current package does not exist in the eval server, then it is created. If evaluation is being done in the editor process and the current package doesn't exist, then the value of `*package*` is used. Normally the package for each file is specified using the `Package` file option (see page 29.)

Set Buffer Package [*Command*]

This command prompts for the name of a package to make the local package in the current buffer. If the current buffer is a slave or background buffer, then the current package in the associated eval server is also changed. In `read eval print` loop interaction, this command should be used instead of `in-package`, since it keeps the editor's notion of the current package up to date.

9.4. 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 (bound to **M-Escape**) [*Command*]

This command prompts for an expression and prints the result of its evaluation in the echo area. If an error happens during evaluation, the evaluation is simply aborted, instead of going into the debugger. This command doesn't return until the evaluation is complete.

Evaluate Defun (bound to **C-x C-e**) [*Command*]
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. The result of the evaluation of each form is displayed in the echo area.

Re-evaluate Defvar [*Command*]

This command is similar to Evaluate Defun. It is used for force the re-evaluation of a **defvar** init form. If the current top-level form is a **defvar**, then it does a **makunbound** on the variable, and evaluates the form.

Compile Defun (bound to **C-x C-c**) [*Command*]
Compile Region [*Command*]

These commands compile the text in the current defun and the region, respectively.

Compile File (bound to **C-x c**) [*Command*]

This command saves the current buffer if it is modified and then compiles the associated file. If a prefix argument is specified, then the user is prompted for a file to compile instead of the one in the current buffer. Since there is a complete log of output in the background 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 ("**.fasl**") file must be loaded.

Compile Group [*Command*]
List Compile Group [*Command*]

Compile Group does a Save All Files and then compiles every "**.lisp**" file for which the corresponding "**.fasl**" file is older or nonexistent. The files are compiled in the order in which they appear in the group definition. A prefix argument forces compilation of all "**.lisp**" files.

List Compile Group lists any files that would be compiled by Compile Group. All Modified files are saved before checking to generate a consistent list.

Set Compile Server [*Command*]
Set Buffer Compile Server [*Command*]

These commands are analogous to Set Eval Server and Set Buffer Eval Server (page 54), but they determine the eval server used for file compilation requests. If a compile server has been specified, then Compile File and Compile Group will send compilation requests to that server instead of the current eval server.

Having a separate compile server makes it easy to do compilations in the background, either in a slave on the local machine, or in a registered eval server on a remote machine.

Edit Compiler Errors [*Command*]

This command provides a convenient way to scan through the compiler errors in a background buffer. After placing the point at the first error message which has not been edited yet, a single character option is prompted for:

Space, y	Skip over this error and go to the next one.
Delete, n	Go back to the preceding error message.
C-r	Go into a recursive edit on the top level form in which the error occurred. The point

is placed at the beginning of the form and the mark at the end. When the recursive edit is exited, the prompting is repeated. If the text has been changed since the compilation, the positioning may be off.

Escape Exit this command.

If this command is called when not in a background buffer, it switches to the background buffer for the current eval server.

9.5. Querying the Environment

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

Describe Function Call (bound to **C-M-A**) [*Command*]
Describe Symbol (bound to **C-M-S**) [*Command*]

Describe Function Call uses the current eval server to describe the symbol found at the head of the currently enclosing list, displaying the output in a pop-up window. Describe Symbol is the same except that it describes the symbol at or before the point. These commands are primarily useful for finding the documentation for functions and variables.

9.6. Editing Definitions

The LISP compiler annotates each compiled function object with the source file that the function was originally defined from. The definition editing commands use this information to locate and edit the source for functions defined in the environment.

Edit Definition [*Command*]
Go to Definition (bound to **C-M-F**) [*Command*]

Edit Definition prompts for the name of a function, and the uses the current eval server to find out which file the function is defined in. If the function isn't defined by a `defun` or `defmacro` form, then just read in the file. If the function is not compiled, then look for it in the current buffer.

Go to Definition is identical, except that it uses the symbol at the beginning of the current list as the function name.

Add Definition Directory Translation [*Command*]
Delete Definition Directory Translation [*Command*]

The defining file is recorded as an absolute pathname. The definition editing commands have a directory translation mechanism that allow the sources to be found when they are not in the location where compilation was originally done. Add Definition Directory Translation prompts for two directory namestrings and causes the first to be mapped to the second. Longer (more specific) directory specifications are matched before shorter (more general) ones.

Delete Definition Directory Translation prompts for a directory namestring and deletes it from the directory translation table.

9.7. Manipulating the Editor Process

When developing Hemlock customizations, it is useful to be able to manipulate the editor LISP environment from Hemlock.

Editor Describe (bound to **Home t, C-_ t**) [*Command*]
This command prompts for an expression, and then evaluates and describes it in the editor process.

Room [*Command*]
Call the `room` function in the editor process, displaying information about allocated storage in a pop-up window.

Load File [*Command*]
Load Pathname Defaults (initial value `nil`) [Hemlock *Variable*]
This command prompts for a file and loads it into the editor process using the the `load`. 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.

9.7.1. Editor Mode

When Editor mode is on, alternate versions of the LISP interaction commands are bound in place of the eval server based commands. These commands manipulate the editor process instead of the current eval server. Turning on editor mode in a buffer allows incremental development of code within the running editor.

Editor Mode [*Command*]
This command turns on Editor minor mode in the current buffer. If it is already on, it is turned off. Editor mode may also be turned on using the `Editor` file option (see page 29.)

Editor Compile Defun (bound to **C-x C-c** in Editor mode) [*Command*]
Editor Compile Region [*Command*]
Editor Evaluate Buffer [*Command*]
Editor Evaluate Defun (bound to **C-x C-e** in Editor mode) [*Command*]
Editor Evaluate Region [*Command*]
Editor Re-evaluate Defvar [*Command*]
Editor Describe Function Call (bound to **C-M-a** in Editor mode) [*Command*]
Editor Describe Symbol (bound to **C-M-s** in Editor mode) [*Command*]
These commands are similar to the standard commands, but modify or examine the LISP process that Hemlock is running in. Terminal I/O is done on the initial window for the editor's LISP process. Output is directed to a pop-up window or the editor's window instead of to the background buffer.

Editor Compile File (bound to **C-x c** in Editor mode) [*Command*]
Editor Compile Group [*Command*]
In addition to compiling in the editor process, these commands differ from the eval server versions in that they direct output to the the Compiler Warnings buffer.

Editor Evaluate Expression (bound to **M-Escape** in Editor mode and **C-M-Escape**) [*Command*]
This command prompts for an expression and evaluates it in the editor process. The results of the evaluation are displayed in the echo area.

9.7.2. Eval Mode

Eval mode is a minor mode that simulates a **`read eval print`** loop running within the editor process. Since LISP program development is usually done in a separate eval server process (see page 53), Eval mode is used primarily for debugging code that must run in the editor process. Eval mode shares some commands with Typescript mode: see section 9.2.

Eval mode doesn't completely support terminal I/O: it rebinds `*standard-output*` to a stream that inserts into the buffer, but `*standard-input*` and `*terminal-io*` are unaffected. This means that if a form being evaluated reads from `*standard-input*` or uses `*terminal-io*`, then the I/O will be done on the original `*terminal-io*` stream for the editor process. Under X windows, this interaction will happen on the editor's initial `xterm` window. When running on a terminal, bad things may happen, since the terminal is already being used for unbuffered character input.

Select Eval Buffer [*Command*]

This command changes to the Eval buffer, creating one if it doesn't already exist. The Eval buffer is created with Lisp as the major mode and Eval and Editor as minor modes.

Eval Input (bound to **Return** in Eval mode) [*Command*]

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 beep if there is not a complete form. Use **Linefeed** to insert line breaks in the middle of a form.

Abort Eval Input (bound to **M-i** in Eval mode) [*Command*]

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

9.7.3. 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 "`go`" will reenter Hemlock and give the "`Debug:`" prompt again.
b	Show a stack backtrace in a pop-up window.
q, Escape	Quit from this error to the nearest command loop.
p	Display a list of the proceed cases and prompt for the number of a proceed-case to proceed. Proceeding may result in prompting in the `Lisp` window.

Only errors within the editor process are handled in this way. Errors during eval server operations are handled using normal terminal I/O on a typescript in the eval server's slave buffer or background buffer (see page 54). Errors due to interaction in a slave buffer will cause the debugger to be entered in the slave buffer.

9.8. Registered Eval Servers

A *registered eval server* is an eval server that has registered itself with the name server, allowing any Hemlock process to manipulate it. Registered eval servers are useful primarily for offloading file compilations onto another machine. A registered eval server is created using the `-register` command line switch (see section 9.9.)

Connect Registered Eval Server [*Command*]

This command creates a connection to a registered eval server. It prompts for the registered name and the local eval server name, creating a corresponding background buffer. Eval servers connected to using this command don't have slave buffers. Any terminal I/O done during an eval server operation will be done on the background buffer's typescript.

9.9. Command Line Switches

Three command line switches control the initialization of editor and eval servers for a LISP process:

-edit [*name*] This switch starts up Hemlock. The optional argument *name* determines the name of the editor server associated with it. *name* defaults to [*machine-name*]**Editor**. If there is a non-switch command line word immediately following the program name, then it is interpreted as a file to edit.

-register [*name*]
This switch causes the LISP process to be established as an eval server with the specified *name*. *name* defaults to [*machine-name*]**Eval**. The name is checked in to the name server so that editor processes may connect to this eval server. If a LISP was started with **-register** on **EXAMPLE.CS.CMU.EDU**, then anyone can connect their editor to the server by using Set Eval Server with the name **[EXAMPLE.CS.CMU.EDU]Eval**.

-slave [*name*] This switch causes the LISP process to become a slave of the editor process *name*. *name* defaults to [*machine-name*]**Editor**. Since the editor can automatically create slaves on its own machine, this switch is useful primarily for creating slaves that run on a different machine. If my machine is **ME.CS.CMU.EDU**, and I want to run a slave on **SLAVE.CS.CMU.EDU**, then I telnet to **slave** and then say:

```
lisp -slave "[ME.CS.CMU.EDU]Editor"
```

Chapter 10

Simple Customization

Hemlock can be 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.

10.1. Keyboard Macros

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

Define Keyboard Macro (bound to **C-x (**) [*Command*]
End Keyboard Macro (bound to **C-x)**) [*Command*]
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 (bound to **C-x e**) [*Command*]
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 (bound to **C-x q**) [*Command*]
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:

Escape 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, and 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 example 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.

10.2. Binding Keys

Bind Key [*Command*]

This command prompts for a command, a key and a kind of binding to make, and then makes 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 makes a key binding which is only in present when that mode is active in the current buffer.

global Makes a global key binding 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.

10.3. Hemlock Variables

A number of commands use Hemlock variables as flags to control their behavior. Often you can get a command to do what you want by setting a variable. Generally the default value for a variable is chosen to be the safest value for novice users.

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.

10.4. Init Files

Hemlock customizations are normally put in the main LISP initialization file, "`init.lisp`", or when compiled "`init.fasl`". 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-M-? when in Lisp mode.
(bind-key "Extract List" '#(#\c-m-?) :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*.

Index

Index