# Hemlock User's Manual

**Bill Chiles**
**Robert A. MacLachlan**

**February 1992**

**CMU-CS-89-133-R1**

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is a revised version of Technical Report CMU-CS-87-158.

## Abstract

This document describes the Hemlock text editor, version M3.2. 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 supported by the Defense Advanced Research Projects Agency (DOD), and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. government.

# 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 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. Key-events

The canonical representation of editor input is a *key-event*. When you type on the keyboard, Hemlock receives key-events. Key-events have names for their basic form, and we refer to this name as a *keysym*. This manual displays keysyms in a **Bold** font. For example, **a** and **b** are the keys that normally cause the editor to insert the characters *a* and *b*.

---

[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.

Key-events have *modifiers* or *bits* indicating a special interpretation of the root key-event. Although the keyboard places limitations on what key-events you can actually type, Hemlock understands arbitrary combinations of the following modifiers: *Control*, *Meta*, *Super*, *Hyper*, *Shift*, and *Lock*. This manual represents the bits in a key-event by prefixing the keysym with combinations of **C-**, **M-**, **S-**, **H-**, **Shift-**, and **Lock**. For example, **a** with both the control and meta bits set appears as **C-M-a**. In general, ignore the shift and lock modifiers since this manual never talks about keysyms that explicitly have these bits set; that is, it may talk about the key-event **A**, but it would never mention **Shift-a**. These are actually distinct key-events, but typical input coercion turns presents Hemlock with the former, not the latter.

Key-event modifiers are totally independent of the keysym. This may be new to you if you are used to thinking in terms of ASCII character codes. For example, with key-events you can distinctly identify both uppercase and lowercase keysyms with the control bit set; therefore, **C-a** and **C-A** may have different meanings to Hemlock.

Some keysyms' names consist of more than a single character, and these usually correspond to the legend on the keyboard. For example, some keyboards let you enter **Home**, **Return**, **F9**, etc.

In addition to a keyboard, you may have a mouse or pointer device. Key-events also represent this kind of input. For example, the down and up transitions of the *left button* correspond to the **Leftdown** and **Leftup** keysyms.

See sections 1.3.1, 1.7, 1.8

### 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 118.

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 sequence of key-events (see section 1.2.1) typed on the keyboard, usually only one or two in length. Sections 1.7 and 1.8 contain information on particular input devices.

When a command is bound to a key, typing the key causes Hemlock to invoke the command. When the command completes its job, Hemlock returns to reading another key, and this continually repeats.

Some commands read key-events interpreting them however each command desires. When commands do this, key bindings have no effect, but you can usually abort Hemlock whenever it is waiting for input by typing **C-g** (see section 1.12). You can usually find out what options are available by typing **C-_** or **Home** (see section 1.10).

The user can easily rebind keys to different commands, bind new keys to commands, or establish bindings for commands never bound before (see section 13.2).

In addition to the key bindings explicitly listed with each command, there are some implicit bindings created by using key translations[2]. These bindings are not displayed by documentation commands such as Where Is. By default, there are only a few key translations. The modifier-prefix characters **C-^**, **Escape**, **C-z**, or **C-c** may be used when typing keys to convert the following key-event to a control, meta, control-meta, or hyper key-event. For example, **C-x Escape b** invokes the same commands as **C-x M-b**, and **C-z u** is the same as **C-M-u**. This allows user to type more interesting keys on limited keyboards that lack control, meta, and hyper keys.

Key Echo Delay (initial value **`1.0`**) [Hemlock *Variable*]

A key binding may be composed of several key-events, especially when you enter it using modifier-prefix key-events. Hemlock provides feedback for partially entered keys by displaying the typed key-events in the echo area. In order to avoid excessive output and clearing of the echo area, this display is delayed by Key Echo Delay seconds. If this variable is set to **`nil`**, then Hemlock foregoes displaying initial subsequences of keys.

### 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-_**.

---

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

## 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.
- 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 control or 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 (initial value **4**) [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 is 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, and these commands use a *pop-up* window to display the information. It is known as a *pop-up* window because it temporarily appears on the screen overlaying text already displayed. Most commands of this nature can generate their output quickly, but in case there is a lot of output, or the user wants to repeatedly refer to the same output while editing, Hemlock saves the output in a buffer. Different commands may use different buffers to save their output, and we refer to these as *random typeout* buffers.

If the amount of output exceeds the size of the pop-up window, Hemlock displays the message "**--More--**" after each window full. The following are valid responses to this prompt:

**Space**, **y** Display the next window full of text.

**Delete**, **Backspace**, **n**
Abort any further output.

**Escape**, **!** Remove the window and continue saving any further output in the buffer.

**k** This is the same as **!** or **escape**, but Hemlock makes a normal window over the pop-up window. This only works on bitmap devices.

Any other input causes the system to abort using the key-event to determine the next command to execute.

When the output is complete, Hemlock displays the string "**--Flush--**" in the pop-up window's modeline, indicating that the user may flush the temporary display. Typing any of the key-events described above removes the pop-up window, but typing **k** still produces a window suitable for normal editing. Any other input also flushes the display, but Hemlock uses the key-event to determine the next command to invoke.

Select Random Typeout Buffer (bound to **H-t**) [*Command*]

This command makes the most recently used random typeout buffer the current buffer in the current window.

Random typeout buffers are always in Fundamental mode.

### 1.6.2. Buffer Display

If a line of text is too long to fit within the screen width it is *wrapped*, with Hemlock displaying consecutive pieces of the text line on as many screen lines as needed to hold the text. Hemlock indicates a wrapped line by placing a line-wrap character in the last column of each screen line. Currently, the line-wrap character is an exclamation point (**!**). It is possible for a line to wrap off the bottom of the screen or on to the top.

Hemlock wraps screen lines when the line is completely full regardless of the line-wrap character. Most editors insert the line-wrap character and wrap a single character when a screen line would be full if the editor had avoided wrapping the line. In this situation, Hemlock would leave the screen line full. This means there are always at least two characters on the next screen line if Hemlock wraps a line of display. When the cursor is at the end of a line which is the full width of the screen, it is displayed in the last column, since it cannot be displayed off the edge.

Hemlock displays most characters as themselves, but it treats some specially:

- Tabs are treated as tabs, with eight character tab-stops.
- Characters corresponding to ASCII control characters are printed as **^***char*; for example, a formfeed is **^L**.
- Characters with the most-significant bit on are displayed as **<***hex-code***>**; for example, **<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, Hemlock always displays the cursor on the first character.

### 1.6.3. Recentering Windows

When redisplaying the current window, Hemlock makes sure the current point is visible. This is the behavior you see when you are entering text near the bottom of the window, and suddenly redisplay shifts your position to the window's center.

Some buffers receive input from streams and other processes, and you might have windows displaying these. However, if those windows are not the current window, the output will run off the bottom of the windows, and you won't be able to see the output as it appears in the buffers. You can change to a window in which you want to track output and invoke the following command to remedy this situation.

Track Buffer Point [*Command*]

This command makes the current window track the buffer's point. This means that each time Hemlock redisplays, it will make sure the buffer's point is visible in the window. This is useful for windows that are not current and that display buffer's that receive output from streams coming from other processes.

### 1.6.4. Modelines

A modeline is the line displayed at the bottom of each window where Hemlock shows information about the buffer displayed in that window. Here is a typical modeline:

```
Hemlock USER: (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

Current Package for Lisp interaction commands is the **"USER"** package. 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 present instead:

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

In this case, the buffer is named Silly and is in Lisp mode. The user has set Current Package for this buffer to **"PLAY"**.

Maximum Modeline Pathname Length (initial value **nil**) [Hemlock *Variable*]

This variable controls how much of a pathname Hemlock displays in a modeline. Some distributed file systems can have very long pathnames which leads to the more particular information in a pathname running off the end of a modeline. When set, the system chops off leading directories until the name is less than the integer value of this variable. Three dots, **...**, indicate a truncated name. The user can establish this variable buffer locally with the Defhvar command.

If the user has modified the buffer since the last time it was read from or save to a file, then the modeline contains an asterisk (*****) between the modes list and the file or buffer name:

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

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

There is a special modeline known as the *status line* which appears as the Echo Area's modeline. Hemlock and user code use this area to display general information not particular to a buffer — recursive edits, whether you just received mail, etc.

## 1.7. Use with X Windows

You should use Hemlock on a workstation with a bitmap display and a windowing system since Hemlock makes good use of a non-ASCII device, mouse, and the extra modifier keys typically associated with workstations. This section discusses using Hemlock under X windows, the only supported windowing system.

### 1.7.1. Window Groups

Hemlock manages windows under X in groups. This allows Hemlock to be more sophisticated in its window management without being rude in the X paradigm of screen usage. With window groups, Hemlock can ignore where the groups are, but within a group, it can maintain the window creation and deletion behavior users expect in editors without any interference from window managers.

Initially there are two groups, a main window and the Echo Area. If you keep a pop-up display, see section 1.6.1, Hemlock puts the window it creates in its own group. There are commands for creating new groups.

Hemlock only links windows within a group for purposes of the Next Window, Previous Window, and Delete Next Window commands. To move between groups, you must use the Point to Here command bound to the mouse.

Window manager commands can reshape and move groups on the screen.

### 1.7.2. Event Translation

Each X key event is translated into a canonical input representation, a key-event. The X key event consists of a scan-code and modifier bits, and these translate to an X keysym. This keysym and the modifier bits map to a key-event.

If you type a key with a shift key held down, this typically maps to a distinct X keysym. For example, the shift of **3** is **#**, and these have different X keysyms. Some keys map to the same X keysym regardless of the shift bit, such as **Tab**, **Space**, **Return**, etc. When the X lock bit is on, the system treats this as a caps-lock, only mapping keysyms for lowercase letters to shifted keysyms.

The key-event has a keysym and a field of bits. The X keysyms map directly to the key-event keysyms. There is a distinct mapping for each CLX modifier bit to a key-event bit. This tends to eliminate shift and lock modifiers, so key-events usually only have control, meta, hyper, and super bits on. Hyper and super usually get turned on with prefix key-events that set them on the following key-event, but you can turn certain keys on the keyboard into hyper and super keys. See the X manuals and the *Hemlock Command Implementor's Manual* for details.

The system also maps mouse input to key-events. Each mouse button has distinct key-event keysyms for whether the user pressed or released it. For convenience, Hemlock makes use of an odd property of converting mouse events to key-events. If you enter a mouse event with the shift key held down, Hemlock sees the key-event keysym for the mouse event, but the key-event has the super bit turned on. For example, if you press the left button with the shift key pressed, Hemlock sees **S-Leftdown**.

Note that with the 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. This is awkward, and it often confuses the X server. For this reason, the commands bound to the middle button are also bound to the shifted left button, **S-Leftdown**, which is much easier to type.

### 1.7.3. 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 being 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.

### 1.7.4. 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.

Thumb Bar Meter (initial value **`t`**) [Hemlock *Variable*]

If this variable is true, then windows will be created to be displayed with a ruler in the bottom border of the window.

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.

Default Initial Window Width (initial value **`80`**) [Hemlock *Variable*]
Default Initial Window Height (initial value **`24`**) [Hemlock *Variable*]
Default Initial Window X [Hemlock *Variable*]
Default Initial Window Y [Hemlock *Variable*]
Default Window Height (initial value **`24`**) [Hemlock *Variable*]
Default Window Width (initial value **`80`**) [Hemlock *Variable*]

Hemlock uses the variables with "Initial" in their names when it first starts up to make its first window. The width and height are specified in character units, but the x and y are specified in pixels. The other variables determine the width and height for interactive window creation, such as making a window with New Window (page 35).

Cursor Bitmap File (initial value **`"library: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.

Default Font [Hemlock *Variable*]

This variable holds the string name of the font to be used for normal text display: buffer text, modelines, random typeout, etc. The font is loaded at initialization time, so this variable must be set before entering Hemlock. When **`nil`**, the display type is used to choose a font.

## 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 Initialization

For best redisplay performance, it is very important to set the terminal speed:

**`stty 2400`**

Often when running Hemlock using TTY redisplay, Hemlock will actually be talking to a PTY whose speed is initialized to infinity. In reality, the terminal will be much slower, resulting in Hemlock's output getting way ahead of the terminal. This prevents Hemlock from briefly stopping redisplay to allow the terminal to catch up. See also Scroll Redraw Ratio (page 10).

The terminal control sequences are obtained from the termcap database using the normal Unix conventions. The **`"TERM"`** environment variable holds the terminal type. The **`"TERMCAP"`** environment variable can be used to override the default termcap database (in **`"/etc/termcap"`**). The size of the terminal can be altered from the termcap default through the use of:

**`stty rows`** *height* **`columns`** *width*

### 1.8.2. Terminal Input

The most important limitation of a terminal is its 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 many more than the 128 key-events available in ASCII.

On a terminal, Hemlock attempts to translate ASCII control characters into the most useful key-event:

- On a terminal, control does not compose with shift. If the control key is down when you type a letter keys, the terminal always sends one code regardless of whether the shift key is held. Since Hemlock primarily binds commands to key-events with keysyms representing lowercase letters regardless of what bits are set in the key-event, the system translates the ASCII control codes to a keysym representing the appropriate lowercase characters. This keysym then forms a key-event with the control bit set. Users can type **C-c** followed by an uppercase character to form a key-event with a keysym representing an uppercase character and bits with the control bit set.
- On a terminal, some of the named keys generate an ASCII control code. For example, **`Return`** usually sends a **`C-m`**. The system translates these ASCII codes to a key-event with an appropriate keysym instead of the keysym named by the character which names the ASCII code. In the above example, typing the **`Return`** key would generate a key-event with the **Return** keysym and no bits. It would NOT translate to a key-event with the **m** keysym and the control bit.

Since terminals have no meta key, you must use the **Escape** and **C-Z** modifier-prefix key-events to invoke commands bound to key-events with the meta bit or meta and control bits set. ASCII terminals cannot generate all key-events which have the control bit on, so you can use the **C-^** modifier-prefix. The **C-c** prefix sets the hyper bit on the next key-event typed.

When running Hemlock from a terminal **`^\`** is the interrupt key-event. 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.

### 1.8.3. Terminal Redisplay

Redisplay is substantially different on a terminal. Hemlock uses different algorithms, and different parameters control redisplay and screen management.

Terminal redisplay uses the Unix termcap database to find out how to use a terminal. Hemlock is useful with terminals that lack capabilities for inserting and deleting lines and characters, and some terminal emulators implement these operations very inefficiently (such as xterm). If you realize poor performance when scrolling, create a termcap entry that excludes these capabilities.

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 from the user, it requests it by displaying a *prompt* in the echo area. The following is a typical prompt:

```
Select Buffer: [hemlock-init.lisp /usr/foo/]
```

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 Hemlock uses if you type **Return** without giving any other input.

There are four general kinds of prompts:

*key-event* The response is a single key-event 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 you only need to supply enough of the keyword to distinguish it from any other choice. In some cases a keyword prompt accepts unknown input, indicating the prompter should create a new entry. If this is the case, then you must enter the keyword fully specified or completed using **Escape**; this distinguishes entering an old keyword from making a new keyword which is a prefix of an old one since the system completes partial input automatically.

*file* The response is the name of a file, which may have to exist. Unlike other prompts, the default has some effect even after the user supplies some input: the system *merges* the default with the input filename. See page 33 for a description of filename merging. **Escape** and **Space** complete the input for a file parse.

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

These key-events 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, Hemlock does two things:

1. beeps, if the variable Beep on Ambiguity set, and
2. moves the point to the end of the first word requiring disambiguation.

This allows you to add to the input before confirming the it again.

**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 or file parse as far as possible, beeping if the result is ambiguous. When the result is ambiguous, Hemlock moves the point to the first ambiguous field, which may be the end of the completed input.

**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, and you can continue entering fields while leaving previous fields ambiguous. For example, you can invoke Forward Word as an extended command by typing **M-X f Space w Return**. Each time the user enters space, Hemlock attempts to complete the current field and all previous fields.

**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 key-event so that it is not interpreted as a command.

Ignore File Types [Hemlock *Variable*]

This variable is a list of file types (or extensions), represented as a string without the dot, e.g. `"fasl"`. Files having any of the specified types will be considered nonexistent for completion purposes, making an unambiguous completion more likely. The initial value contains most common binary and output file types.

## 1.10. Online Help

Hemlock has a fairly good online documentation facility. You can get brief documentation for every command, variable, character attribute, and key by typing a key.

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

This command prompt for a key-event indicating one of a number of other documentation commands. The following are valid responses:

| | |
|---|---|
| **a** | List commands and other things whose names contain a specified keyword. |
| **d** | Give the documentation and bindings for a specified command. |
| **g** | Give the documentation for any Hemlock thing. |
| **v** | Give the documentation for a Hemlock variable and its values. |
| **c** | Give the documentation for a command bound to some key. |
| **l** | List the last sixty key-events typed. |
| **m** | Give the documentation for a mode followed by a short description of its mode-specific bindings. |
| **p** | Give the documentation and bindings for commands that have at least one binding involving a mouse/pointer key-event. |
| **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. |

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

This command prints brief documentation for all commands, variables, and character attributes whose names match the input. This performs a prefix match on each supplied word separately, intersecting the names in each word's result. For example, giving Apropos "`f m`" causes it to tersely describe following commands and variables:

- Auto Fill Mode
- Fundamental Mode
- Mark Form
- Default Modeline Fields
- Fill Mode Hook
- Fundamental Mode Hook

Notice Mark Form demonstrates that the "`f`" words may follow the "`m`" order of the fields does not matter for Apropos.

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 and all the keys bound to it.

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.

Describe Mode (bound to **Home m, C-_ m**) [*Command*]

This command prints the documentation for a mode followed by a short description of each of its mode-specific bindings.

Show Variable [*Command*]
Describe and Show Variable [*Command*]

Show Variable prompts for the name of a variable and displays the global value of the variable, the value local to the current buffer (if any), and the value of the variable in all defined modes that have it as a local variable. Describe and Show Variable displays the variable's documentation in addition to the values.

What Lossage (bound to **Home l, C-_ l**) [*Command*]

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

Describe Pointer [*Command*]

This command displays the documentation and bindings for commands that have some binding involving a mouse/pointer key-event. It will not show the documentation for the Illegal command regardless of whether it has a pointer binding.

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 is available is displayed.

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 82.

**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 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 key-events that they want as input. If you get in such a situation and want to get out, you can usually 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 re-enter it.
- 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 Hemlock will save all modified buffers with associated files.
- If you lose changes to a file due to a crash or accidental failure to save, look for backup ("*file*.**BAK**") or checkpoint ("*file*.**CKP**") files in the same directory where the file was.
- If the screen changes unexpectedly, you may have accidentally typed an incorrect command. Use **Home l** 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 these commands refrain from asking, no matter how large the region is.

Undo [*Command*]

This command undoes the last major modification. Killing commands and some other commands save information about their modifications, so accidental uses may be retracted. This command displays the name of the operation to be undone and asks for confirmation. If the affected text has been modified between the invocations of Undo and the command to be undone, then the result may be somewhat incorrect but useful. Often Undo itself can be undone by invoking it again.

## 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 user exits a recursive edit, the command that entered it proceeds as though nothing happened. Hemlock notes recursive edits in the Echo Area modeline, or status line. A counter reflects the number of pending recursive edits.

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 this signals an user error.

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

This command causes the command which invoked the recursive edit to get an error. If not in a recursive edit, this signals an user error.

## 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 save modified buffers in a seriously broken Hemlock. To use this, type "**`(save-all-buffers)`**" to the top-level ("**`* `**") or debugger ("**`1] `**") prompt and confirm saving of each buffer that should be saved. 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*]
Goto Absolute Line [*Command*]

Next Line and Previous Line 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.

Goto Absolute Line moves to the indicated line, as if you counted them starting at the beginning of the buffer with number one. If the user supplies a prefix argument, it is the line number; otherwise, Hemlock prompts the user for 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 screenful of text, the exact amount being determined by the size of the window. If a prefix argument is supplied, then this scrolls the screen that many lines. When this action scrolls the line with the point off the screen, it this command moves the point to the vertical center of the window. 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 18).

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. To use these commands, one must first use some command to mark the region.

Although the mark is always pointing somewhere (initially to the beginning of the buffer), region commands insist that the region be made *active* before it can be used. This prevents accidental use of a region command from mysteriously mangling large amounts of text.

Active Regions Enabled (initial value `t`) [Hemlock *Variable*]

When this variable is true, region commands beep unless the region is active. This may be set to `nil` for more traditional Emacs region semantics.

Once a marking command makes the region active, it remains active until:

- a command uses the region,
- a command modifies the buffer,
- a command changes the current window or buffer,
- a command signals an editor error,
- or the user types **C-g**.

Motion commands have the effect of redefining the region, since they move the point and leave the region active.

Commands that insert a large chunk of text into the buffer usually set an *ephemerally active* region around the inserted text. An ephemerally active region is always deactivated by the next command, regardless of the kind of command. The ephemerally active region allows an immediately following region command to manipulate the inserted text, but doesn't persist annoyingly. This is also very useful with active region highlighting, since it visibly marks the inserted text.

Highlight Active Region (initial value **t**) [Hemlock *Variable*]
Active Region Highlighting Font (initial value **nil**) [Hemlock *Variable*]

When Highlight Active Region is true, Hemlock displays the text in the region in a different font whenever the region is active. This provides a visible indication of what text will be manipulated by a region command. Active region highlighting is only supported under X windows.

Active Region Highlighting Font is the name of the font to use for active region highlighting. If unspecified, Hemlock uses an underline font.

Set/Pop Mark (bound to **C-@**) [*Command*]

This command moves the mark to the point (saving the old mark on the mark stack) and activates the region. After using this command to mark one end of the region, use motion commands to move to the other end, then do the region command. This is the traditional Emacs marking command; when running under a windowing system with mouse support, it is usually easier to use the mouse with the Point to Here (page 20) and Generic Pointer Up (page 20).

For historical reasons, the prefix argument causes this command to do things that are distinct commands in Hemlock. A prefix argument of four does Pop and Goto Mark, and a prefix argument of **16** does Pop Mark.

Mark Whole Buffer (bound to **C-x h**) [*Command*]
Mark to Beginning of Buffer (bound to **C-<**) [*Command*]
Mark to End of Buffer (bound to **C->**) [*Command*]

Mark Whole Buffer 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.

Mark to Beginning of Buffer sets the current region from point to the beginning of the buffer.

Mark to End of Buffer sets the current region from the end of the buffer to point.

Activate Region (bound to **C-x C-Space, C-x C-@**) [*Command*]

This command makes the region active, using whatever the current position of the mark happens to be. This is useful primarily when the region is accidentally deactivated.

### 2.2.1. The Mark Stack

As was hinted at earlier, each buffer has a *mark stack*, providing a history of positions in that buffer. The current mark is the mark on the top of the stack; earlier values are recovered by popping the stack. Since commands that move a long distance save the old position on the mark stack, the mark stack commands are useful for jumping to interesting places in a buffer without having to do a search.

Pop Mark (bound to **C-M-Space**) [*Command*]
Pop and Goto Mark (bound to **M-@, M-Space**) [*Command*]

Pop Mark pops the mark stack, restoring the current mark to the next most recent value. Pop and Goto Mark also pops the mark stack, but instead of discarding the current mark, it moves the point to that position. Both commands deactivate the region.

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

This command 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. The old mark isn't pushed on the mark stack, since it is saved in the point.

### 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 7.)

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*]

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

When used in a window's modeline, this moves the point of the window's buffer to the position within the file that is the same percentage, start to end, as the horizontal position of the mouse within the modeline. This also makes this window current if necessary.

This command supplies a function Generic Pointer Up invokes if it runs without any intervening generic pointer up predecessors executing. If the position of the pointer is different than the current point when the user invokes Generic Pointer Up, then this function pushes a buffer mark at point and moves point to the pointer's position. This allows the user to mark off a region with the mouse.

Generic Pointer Up (bound to **Middleup, S-Leftup**) [*Command*]

Other commands determine this command's action by supplying functions that this command invokes. The following built-in commands supply the following generic up actions:

Point to Here
: When the position of the pointer is different than the current point, the action pushes a buffer mark at point and moves point to the pointer's position.

Bufed Goto and Quit
: The action is a no-op.

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

This command is a combination of Point to Here and Un-Kill (page 22). 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, you can insert characters with graphic representations by typing the corresponding key-event which you normally generate with the obvious keyboard key. You can only insert characters whose codes correspond to ASCII codes. To insert those without graphic representations, use Quoted Insert.

Self Insert [*Command*]

Self Insert inserts into the buffer the character corresponding to the key-event typed to invoke the command. This command is normally bound to all such key-events **Space**. If a prefix argument is supplied, then this inserts the character 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 key-events have corresponding ASCII characters, but these key-events are bound to commands other than Self Insert. Sometimes they are otherwise encumbered such as with **C-g**. Quoted Insert prompts for a key-event, without any command interpretation semantics, and inserts the corresponding character. If the appropriate character has some code other than an ASCII code, this will beep and abort the command. A common use for this command is inserting a **Formfeed** by typing **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. See also Delete Blank Lines (page 24).

### 2.3.2. Deleting Characters

There are a number of commands for deleting characters as well.

Character Deletion Threshold (initial value **5**) [Hemlock *Variable*]

If more than this many characters are deleted by a character deletion command, then the deleted text is placed in the kill ring.

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 a 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.

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

This command transposes two regions with endpoints defined by the mark stack and point. To use this command, place three marks (in order) at the start and end of the first region, and at the start of the second region, then place the point at the end of the second region. Unlike the other transposition commands, a second use will simply undo the effect of the first use, and to do even this, you must reactivate the current region.

### 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.

Delete Blank Lines (bound to **C-x C-o**) [*Command*]

This command deletes all blank lines surrounding the current line, leaving the point on a single blank line. If the point is already on a single blank line, then that line is deleted. If the point is on a non-blank line, then all blank lines immediately following that line are deleted. This command is often used to clean up after Open Line (page 21).

## 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.

String Search Ignore Case (initial value **t**) [Hemlock *Variable*]

This variable determines the kind of search done by searching and replacing commands.

Incremental Search (bound to **C-s**) [*Command*]
Reverse Incremental Search (bound to **C-r**) [*Command*]

Incremental Search searches for an occurrence of a string after the current point. It is known as an incremental search because it reads key-events form the keyboard one at a time and immediately searches for the pattern of corresponding characters as you type. 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.

This command dispatches on the following key-events as sub-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 key-event typed. If that key-event simply added to the search pattern, then this removes the character from the pattern, moving back to the last match found before entering the removed character. If the character was a **C-s** or **C-r**, then this moves back to the previous match and possibly reverses the search direction.

**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 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 character corresponding to the next key-event, rather than treating it as a command.

Any key-event not corresponding to a graphic character, except those just described, causes the search to exit. Hemlock then uses the key-event in it normal command interpretation.

For example, typing **C-a** will exit the search *and* go to the beginning of the current line. When either of these commands successfully exits, they push the starting position (before the search) on the mark stack. If the current region was active when the search started, this foregoes pushing a mark.

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. When either of these commands successfully exits, they push the starting position (before the search) on the mark stack. This is inhibited when the current region is active.

Query Replace (bound to **M-%**) [*Command*]

This command prompts in the echo area for a target string and a replacement string. It then searches for an occurrence of the target after the point. When it finds a match, it prompts for a key-event indicating what action to take. The following are valid responses:

**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 14) 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 key-event causes the command to exit, returning the key-event to the input stream; thus, Hemlock will interpret it normally for a command binding.

When the current region is active, this command uses it instead of the region from point to the end of the buffer. This is especially useful when you expect to use the **!** option.

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. On a successful exit from this command, the starting position (before the search) is pushed on the mark stack.

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 it operates without ever querying the user before making replacements. After prompting for a target and replacement string, it replaces all occurrences of the target string following the point. If a prefix argument is specified, then only that many occurrences are replaced. When the current region is active, this command uses it instead of the region from point to the end of the buffer.

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 this displays that many lines before and after each matching line. When the current region is active, this command uses it instead of the region from point to the end of the buffer.

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. When the current region is active, these commands uses it instead of the region from point to the end of the buffer.

## 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**'s.) The first non-blank line after the page marker is the *page title*. The page commands are quite useful when logically distinct parts of a file are put on separate pages. See also Count Lines Page (page 28). These commands only recognize **^L**'s at the beginning of a lines, so those quoted in string literals do not get in the way.

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 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. Repeated invocations in this manner continue searching from the point of the last find, and a first search with a particular pattern pushes a buffer mark.

*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. With a prefix argument, Insert Page Directory inserts at the point.

## 2.7. Counting Commands

Count Words [*Command*]

This command counts the number of words from the current point to the end of the buffer, displaying a message in the echo area. When the current region is active, this uses it instead of the region from the point to the end of the buffer. Word delimiters are determined by the current major mode.

Count Lines [*Command*]

This command counts the number of lines from the current point to the end of the buffer, displaying a message in the echo area. When the current region is active, this uses it instead of the region from the point to the end of the buffer.

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 that string in the text from the point to the end of the buffer. When the current region is active, this uses it instead of the region from the point to the end of the buffer.

## 2.8. Registers

Registers allow you to save a text position or chunk of text associated with a key-event. This is a convenient way to repeatedly access a commonly-used location or text fragment. The concept and key bindings should be familiar to TECO users.

Save Position (bound to **C-x s**) [*Command*]
Jump to Saved Position (bound to **C-x j**) [*Command*]

These commands manipulate registers containing textual positions. Save Position prompts for a register and saves the location of the current point in that register. Jump to Saved Position prompts for a register and moves the point to the position saved in that register. If the saved position is in a different buffer, then that buffer is made current.

Put Register (bound to **C-x x**) [*Command*]
Get Register (bound to **C-x g**) [*Command*]

These commands manipulate registers containing text. Put Register prompts for a register and puts the text in the current region into the register. Get Register prompts for a register and inserts the text in that register at the current point.

List Registers [*Command*]
Kill Register [*Command*]

List Registers displays a list of all the currently defined registers in a pop-up window, along with a brief description of their contents. Kill Register prompts for the name of a register and deletes that register.

# 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. A buffer with no associated file is represented by the buffer name followed by the number of lines in the buffer. A buffer with an associated file are is represented by the name and type of the file, a space, and the device and directory. If the buffer name doesn't match the associated file, then the buffer name is also displayed. When given a prefix argument, this command lists 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*. This is primarily useful in cases where a user accidentally modifies a buffer and then undoes the change. Resetting the modified flag 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.

Set Buffer Writable [*Command*]

This command ensures the current buffer is modifiable.

Insert Buffer [*Command*]

This command prompts for the name of a buffer and inserts its contents at the point, pushing a buffer mark before inserting. 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 key-event to indicate what action to take. It recognizes the following key-events:

**Return, Space, y**
: Prompt for a file in which to save the current buffer and then read in the file found to be modified on disk.

**Delete, Backspace, n**
: Forego reading the file.

**r**
: Read the file found to be modified on disk into the buffer containing the earlier version with modifications. This loses all changes you had in the buffer.

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. This command warns if some other buffer also contains the file.

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 the 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 that 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 or the checkpoint file for that file, whichever is more recent. The point is put in approximately the same place that it was before the file was read. If the original file is reverted to, then clear the modified flag, otherwise leave it set. If a prefix argument is specified, then always revert to the original file, ignoring any checkpoint file.

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, pushing a buffer mark before inserting.

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 some file writing commands add a newline at the end of the file if the last line is non-empty.

**`:ask-user`** Ask the user whether to add a newline.

**`t`** Automatically add a newline and inform the user.

**`nil`** Never add a newline and do not ask.

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.

Keep Backup Files (initial value **`nil`**) [Hemlock *Variable*]

Whenever a file is written by Save File and similar commands, the old file is renamed by appending "**`.BAK`**" to the name, ensuring that some version of the file will survive a system crash during the write. If set to true, this backup file will not deleted even when the write successfully completes.

### 3.3.1. Auto Save Mode

Save mode protects against loss of work in system crashes by periodically saving modified buffers in checkpoint files.

Auto Save Mode [*Command*]

This command turns on Save mode if it is not on, and turns off when it is on. Save mode is on by default.

Auto Save Checkpoint Frequency (initial value **`120`**) [Hemlock *Variable*]
Auto Save Key Count Threshold (initial value **`256`**) [Hemlock *Variable*]

These variables determine how often modified buffers in Save mode will be checkpointed. Checkpointing is done after Auto Save Checkpoint Frequency seconds, or after Auto Save Key Count Threshold keystrokes that modify the buffer (whichever comes first). Either kind of checkpointing may be disabled by setting the corresponding variable to **`nil`**.

Auto Save Cleanup Checkpoints (initial value **`t`**) [Hemlock *Variable*]

If this variable is true, then any checkpoint file for a buffer will be deleted when the buffer is successfully saved in its associated file.

Auto Save Filename Pattern (initial value **`"~A~A.CKP"`**) [Hemlock *Variable*]
Auto Save Pathname Hook (initial value **`make-unique-save-pathname`**) [Hemlock *Variable*]

These variables determine the naming of checkpoint files. Auto Save Filename Pattern is a format string used to name the checkpoint files for buffers with associated files. Format is called with two arguments: the directory and file namestrings of the associated file.

Auto Save Pathname Hook is a function called by Save mode to get a checkpoint pathname when there is no pathname associated with a buffer. It should take a buffer as its argument and return either a pathname or **`nil`**. If a pathname is returned, then it is used as the name of the checkpoint file. If the function returns **`nil`**, or if the hook variable is **`nil`**, then Save mode is turned off in the buffer. The default value for this variable returns a pathname in the default directory of the form "**`save-`***number*", where *number* is a number used to make the file unique.

### 3.3.2. Filename Defaulting and Merging

When Hemlock prompts for the name of a file, it always offers a default. Except for a few commands that have their own defaults, filename defaults are computed in a standard way. If it exists, the associated file for the current buffer is used as the default, otherwise a more complex mechanism creates a default.

Pathname Defaults (initial value **`(pathname "gazonk.del")`**) [Hemlock *Variable*]
Last Resort Pathname Defaults Function [Hemlock *Variable*]
Last Resort Pathname Defaults (initial value **`(pathname "gazonk")`**) [Hemlock *Variable*]

These variables control the computation of default filename defaults when the current buffer has no associated file.

Pathname Defaults holds a "sticky" filename default. Commands that prompt for files set this to the file specified, and the value is used as a basis for filename defaults. It is undesirable to offer the unmodified value as a default, since it is usually the name of an existing file that we don't want to overwrite. If the current buffer's name is all alphanumeric, then the default is computed by substituting the buffer name for the the name portion of Pathname Defaults. Otherwise, the default is computed by calling Last Resort Pathname Defaults Function with the buffer as an argument.

The default value of Last Resort Pathname Defaults Function merges Last Resort Pathname Defaults with Pathname Defaults. Unlike Pathname Defaults, Last Resort Pathname Defaults is not modified by file commands, so setting it to a silly name ensures that real files aren't inappropriately offered as defaults.

When a default is present in the prompt for a file, Hemlock *merges* the given input with the default filename. The semantics of merging, described in the Common Lisp manual, is somewhat involved, but Hemlock has a few rules it uses:

1. If Hemlock can find the user's input as a file on the **`"default:"`** search list, then it forgoes merging with the displayed default. Basically, the system favors the files in your current working directory over those found by merging with the defaults offered in the prompt.

2. Merging comes in two flavors, just merge with the displayed default's directory or just merge with the displayed default's **`file-namestring`**. If the user only responds with a directory specification, without any name or type information, then Hemlock merges the default's **`file-namestring`**. If the user responds with any name or type information, then Hemlock only merges with the default's directory. Specifying relative directories in this second situation coordinates with the displayed defaults, not the current working directory.

### 3.3.3. Type Hooks and File Options

When a file is read either by Find File or Visit File, Hemlock attempts to guess the correct mode in which to put the buffer, 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*.

The user specifies file options with a special syntax on the first line of a file. If the first line contains the string "**`-*-`**", then Hemlock interprets the text between the first such occurrence and the second, which must be contained in one line , as a list of "*option*`:` *value*" pairs separated by semicolons. The following is a typical example:

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

These options are currently defined:

| | |
|---|---|
| Dictionary | The argument is the filename of a spelling dictionary associated with this file. The handler for this option merges the argument with the name of this file. See Set Buffer Spelling Dictionary (page 41). |
| Log | The argument is the name of the change log file associated with this file (see page 47). The handler for this option merges the argument with the name of this file. |
| Mode | The argument is a comma-separated list of the names of modes to turn on in the buffer that the file is read into. |
| 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 75.) |
| Editor | The handler for this option ignores its argument and turns on Editor mode (see Editor Mode (page 81)). |

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.

## 3.4. Windows

Hemlock windows display a portion of a buffer's text. See the section on *window groups*, 1.7.1, for a discussion of managing windows on bitmap device.

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

This command prompts users for a new window which they can place anywhere on the screen. This window is in its own group. This only works with bitmap devices.

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

This command splits the current window roughly in half to make two windows. If the current window is too small to be split, the command signals a user error.

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.

On bitmap devices, if there is only one window in the group, either command deletes the group, making some window in another group the current window. If there are no other groups, they signal a user error.

Go to One Window [*Command*]

This command deletes all window groups leaving one with the Default Initial Window X, Default Initial Window Y, Default Initial Window Width, and Default Initial Window Height. This remaining window retains the contents of the current window.

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, centering the current window about the current line. When the user supplies a positive argument, it scrolls that line to the top of the window. When the argument is negative, the line that far from the bottom of the window is moved to the bottom of the window. In either case when an argument is supplied, this command only refreshes 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 are 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 key-event to determine which Scribe directive to insert. Directives are inserted differently depending on their kind:

*environment* — The current or next paragraph is enclosed in a begin-end pair: **`@begin[`***directive***`]`** *paragraph* **`@end[`***directive***`]`**. If the current region is active, then this command encloses the region instead of the paragraph it would otherwise chose.

*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 key-events on which it dispatches.

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 key-event on which to dispatch.

Add Scribe Paragraph Delimiter [*Command*]
List Scribe Paragraph Delimiters [*Command*]

Add Scribe Paragraph Delimiter prompts for a string to add to the list of formatting commands that delimit paragraphs in Scribe mode. If the user supplies a prefix argument, then this command removes the string as a delimiter.

List Scribe Paragraph Delimiters displays in a pop-up window the Scribe commands that delimit paragraphs.

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 Insert Bracket [*Command*]
Scribe Bracket Table [Hemlock *Variable*]

Scribe Insert Bracket inserts a bracket (**>**, **}**, **)**, or **]**), that caused its invocation, and then shows the matching bracket.

Scribe Bracket Table 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 of two characters.

Check 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 invoking this command:

1. This command displays the message "**Found it.**" in the echo area. This means it found the word in the dictionary exactly as given.
2. This command displays the message "**Found it because of** *word***.**", 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 command prompts with "**Correction choice:**" in the echo area and lists possible correct spellings associated with numbers in a pop-up display. Typing a number selects the corresponding correction, and the command replaces the erroneous word, preserving case as though by Query Replace. Typing anything else rejects all the choices.
4. This commands displays the message "**Word not found.**". The word is not in the dictionary and possibly spelled correctly anyway. Furthermore, no similarly spelled words were found to offer as possible corrections. If this happens, it is worth trying some alternate spellings since one of them might be close enough to some known words that this command could display.

Correct Buffer Spelling [*Command*]

This command scans the entire buffer looking for misspelled words and offers to correct them. It creates a window into the Spell Corrections buffer, and in this buffer it maintains a log of any actions taken by the user. When this finds an unknown word, it prompts for a key-event. The user has the following options:

**a** Ignore this word. If the command finds the word again, it will prompt again.

**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 command will make the same correction automatically, leaving a note in the log window.

**r** Prompt for a word to use instead of the misspelled one, remembering the correction as with **c**.

**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 leaving the buffer around for future reference.

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

If this variable is true, then Auto 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.

Remove Word from Spelling Dictionary [*Command*]

This command prompts for a word to remove from the spelling dictionary. Due to the dictionary representation, removal of a word in the initial spelling dictionary will remove all words with the same root. The user is asked for confirmation before removing a root word with valid suffix flags.

List Incremental Spelling Insertions [*Command*]

This command displays the incremental spelling insertions for the current buffer's associated spelling dictionary file.

Read 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.

Save Incremental Spelling Insertions [*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. The file is prompted for unless Set Buffer Spelling Dictionary has been executed in the buffer.

Set Buffer Spelling Dictionary [*Command*]

This command Prompts for the dictionary file to associate with the current buffer. If the specified dictionary file has not been read for any other buffer, then it is read. Incremental spelling insertions from this buffer can be appended to this file with Save Incremental Spelling Insertions. If a buffer has an associated spelling dictionary, then saving the buffer's associated file also saves any incremental dictionary insertions. The "**`Dictionary:`** *file*" file option may also be used to specify the dictionary for a buffer (see section 3.3.3).

Default User Spelling Dictionary (initial value **`nil`**) [Hemlock *Variable*]

This variable holds the pathname of a dictionary to read the first time Spell mode is entered in a given editing session. When Set Buffer Spelling Dictionary or the "**`dictionary`**" file option is used to specify a dictionary, this default one is read also. It defaults to nil.

### 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.

Auto 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 previously supplied by the user, then this command corrects the spelling. If there is no correction, then this displays a message in the echo area indicating the word is unknown. An unknown word detected by this command may be corrected using the Correct Last Misspelled Word command. This command executes in addition to others bound to the same key; for example, if Fill mode is on, any of its commands bound to the same keys as this command also run.

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*]
Spelling Un-Correct Prompt for Insert (initial value **`nil`**) [Hemlock *Variable*]

Undo Last Spelling Correction 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. When Spelling Un-Correct Prompt for Insert is true, the user is asked to confirm the insertion into the dictionary.

Correct Last Misspelled Word (bound to **M-:**) [*Command*]

This command places the cursor after the last misspelled word detected by the Auto Check Word Spelling command and then prompts for a key-event on which it dispatches:

**c** Display possible corrections in a pop-up window, and prompt for the one to make according to the corresponding displayed digit or letter.

*any digit* Similar to **c** *digit*, but immediately makes the correction, dispensing with display of the possible corrections. This is shorter, but only works when there are less than ten 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. Page 77 describes the Compile Group command.

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 target and replacement strings and then executes an interactive string replace on each file in the active group. This reads in each file as if Find File were used and processes it as if Query Replace were executing.

Group Replace [*Command*]

This is like Group Query Replace except that it executes 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. This reads in each file as if Find File were used. When it finds an occurrence, it prompts the user for a key-event indicating 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*]

The group searching and replacing commands read each file into its own buffer using Find File. Since this may result in large amounts of memory being consumed by unwanted buffers, this variable controls whether to delete the buffer after processing it. When this variable is false, the default, the commands delete the buffer if it did not previously exist; however, regardless of this variable, if the user leaves the buffer modified, the commands will not delete it.

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

If this variable is true, the group searching and replacing commands ask for confirmation before saving any modified file. The commands attempt to save each file processed before going on to the next one in the group.

## 5.2. Source Comparison

These 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.

Source Compare Default Destination (initial value `"Differences"`) [Hemlock *Variable*]

This is a sticky default buffer name to offer when comparison commands prompt for a buffer in which to insert the results.

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.

Buffer Changes [*Command*]

This command compares the contents of the current buffer with the disk version of the associated file. It reads the file into the buffer Buffer Changes File, and generates the comparison in the buffer Buffer Changes Result. As with Compare Buffers, the output buffer is displayed in the current window.

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 being compared is generated in the output buffer. This copies text that is identical in the two comparison buffers to the output buffer. When it encounters a difference, it displays the two differing sections in the output buffer and prompts the user for a key-event indicating what action to take. The following 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 differing sections are too complex, or it is unclear which is the correct version. If you cannot make the decision conveniently at this point, you can later search for the marking string above.

**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 non-**`nil`**, Compare Buffers and Merge Buffers will do comparisons case-insensitively.

Source Compare Ignore Indentation (initial value **`nil`**) [Hemlock *Variable*]

If this variable is non-**`nil`**, Compare Buffers and Merge Buffers ignore initial whitespace when comparing lines.

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 34.)

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 14) 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. Dired Mode

Hemlock provides a directory editing mechanism. The user can flag files and directories for deletion, undelete flagged files, and with a keystroke read in files and descend into directories. In some implementations, it also supports copying, renaming, and a simple wildcard feature.

### 6.1.1. Inspecting Directories

Dired (bound to **C-x C-M-d**) [*Command*]

This command prompts for a directory and fills a buffer with a verbose listing of that directory. When the prefix argument is supplied, this includes Unix dot files. If a dired buffer already exists for the directory, this switches to the buffer and makes sure it displays dot files if appropriate.

Dired with Pattern (bound to **C-x C-M-d**) [*Command*]

This command prompts for a directory and a pattern that may contain at most one wildcard, an asterisk, and it fills a buffer with a verbose listing of the files in the directory matching the pattern. When the prefix argument is supplied, this includes Unix dot files. If a dired buffer already exists for this directory, this switches to the buffer and makes sure it displays dot files if appropriate.

Dired from Buffer Pathname [*Command*]

This command invokes Dired on the directory name of the current buffer's pathname.

Dired Help (bound to **Dired: ?**) [*Command*]

This command pops up a help window listing the various Dired commands.

Dired View File (bound to **Dired: Space**) [*Command*]
Dired Edit File (bound to **Dired: e**) [*Command*]

These command read in the file on the current line with the point. If the line describes a directory instead of a file, then this command effectively invokes Dired on the specification. This associates the file's buffer with the Dired buffer.

Dired View File reads in the file as if by View File, and Dired Edit File as if by Find File.

Dired View File always reads into a newly created buffer, warning if the file already exists in some buffer.

Dired Up Directory (bound to **Dired: ^**) [*Command*]

This command invokes Dired on the directory up one level from the current Dired buffer. This is useful for going backwards after repeatedly invoking Dired View File and descending into a series of subdirectories. Remember, Dired only generates directory listings when no buffer contains a dired for the specified directory.

Dired Update Buffer (bound to **Dired: H-u**) [*Command*]

This command is useful when the user knows the directory in the current Dired buffer has changed. Hemlock cannot know the directory structure has changed, but the user can explicitly update the buffer with this command instead of having to delete it and invoke Dired again.

Dired Next File [*Command*]
Dired Previous File [*Command*]

These commands move to next or previous undeleted file.

### 6.1.2. Deleting Files

Dired Delete File and Down Line (bound to **Dired: d**) [*Command*]

This command marks for deletion the file on the current line with the point and moves point down a line.

Dired Delete File with Pattern (bound to **Dired: D**) [*Command*]

This command prompts for a name pattern that may contain at most one wildcard, an asterisk, and marks for deletion all the names matching the pattern.

Dired Delete File (bound to **Dired: C-d**) [*Command*]

This command marks for deletion the file on the current line with the point without moving the point.

### 6.1.3. Undeleting Files

Dired Undelete File and Down Line (bound to **Dired: u**) [*Command*]

This command unmarks for deletion the file on the current line with the point and moves point down a line.

Dired Undelete File with Pattern (bound to **Dired: U**) [*Command*]

This command prompts for a name pattern that may contain at most one wildcard, an asterisk, and unmarks for deletion all the names matching the pattern.

Dired Undelete File (bound to **Dired: C-u**) [*Command*]

This command unmarks for deletion the file on the current line with the point without moving the point.

### 6.1.4. Expunging and Quitting

Dired Expunge Files (bound to **Dired: !**) [*Command*]
Dired File Expunge Confirm (initial value **`t`**) [Hemlock *Variable*]
Dired Directory Expunge Confirm (initial value **`t`**) [Hemlock *Variable*]

This command deletes files marked for deletion, asking the user for confirmation once for all the files flagged. It recursively deletes any marked directories, asking the user for confirmation once for all those marked. Dired File Expunge Confirm and Dired Directory Expunge Confirm when set to **`nil`** individually inhibit the confirmation prompting for the appropriate deleting.

Dired Quit (bound to **Dired: q**) [*Command*]

This command expunges any marked files or directories as if by Expunge Dired Files before deleting the Dired buffer.

### 6.1.5. Copying Files

Dired Copy File (bound to **Dired: c**) [*Command*]

This command prompts for a destination specification and copies the file on the line with the point. When prompting, the current line's specification is the default, which provides some convenience in supplying the destination. The destination is either a directory specification or a file name, and when it is the former, the source is copied into the directory under its current file name and extension.

Dired Copy with Wildcard (bound to **Dired: C**) [*Command*]

This command prompts for a name pattern that may contain at most one wildcard, an asterisk, and copies all the names matching the pattern. When prompting for a destination, this provides the Dired buffer's directory as a default. The destination is either a directory specification or a file name with a wildcard. When it is the former, all the source files are copied into the directory under their current file names and extensions. When it is the later, each sources file's substitution for the wildcard causing it to match the first pattern replaces the wildcard in the destination pattern; for example, you might want to copy **`"*.txt"`** to **`"*.text"`**.

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

This variable controls interaction with the user when it is not obvious what the copying process should do. This takes one of the following values:

**`t`** When the destination specification exists, the copying process stops and asks the user if it should overwrite the destination.

**`nil`** The copying process always copies the source file to the destination specification without interacting with the user.

**`:update`** When the destination specification exists, and its write date is newer than the source's write date, then the copying process stops and asks the user if it should overwrite the destination.

### 6.1.6. Renaming Files

Dired Rename File (bound to **Dired: r**) [*Command*]

Rename the file or directory under the point

Dired Rename with Wildcard (bound to **Dired: R**) [*Command*]

Rename files that match a pattern containing ONE wildcard.

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

When non-nil, Dired will query before clobbering an existing file.

## 6.2. View Mode

View mode provides for scrolling through a file read-only, terminating the buffer upon reaching the end.

View File [*Command*]

This command reads a file into a new buffer as if by "Visit File", but read-only. Bindings exist for scrolling and backing up in a single key stroke.

View Help (bound to **View: ?**) [*Command*]

This command shows a help message for View mode.

View Edit File (bound to **View: e**) [*Command*]

This commands makes a buffer in View mode a normal editing buffer, warning if the file exists in another buffer simultaneously.

View Scroll Down (bound to **View: Space**) [*Command*]
View Scroll Deleting Buffer (initial value **`t`**) [Hemlock *Variable*]

This command scrolls the current window down through its buffer. If the end of the file is visible, then this deletes the buffer if View Scroll Deleting Buffer is set. If the buffer is associated with a Dired buffer, this returns there instead of to the previous buffer.

View Return (bound to **View: ^**) [*Command*]
View Quit (bound to **View: q**) [*Command*]

These commands invoke a function that returns to the buffer that created the current buffer in View mode. Sometimes this function does nothing, but it is useful for returning to Dired buffers and similar Hemlock features.

After invoking the viewing return function if there is one, View Quit deletes the buffer that is current when the user invokes it.

Also, bound in View mode are the following commands:

**backspace**, **delete**
Scrolls the window up.

**<** Goes to the beginning of the buffer.

**>** Goes to the end of the buffer.

## 6.3. Process Mode

Process mode allows the user to execute a Unix process within a Hemlock buffer. These commands and default bindings cater to running Unix shells in buffers. For example, Stop Buffer Subprocess is bound to **H-z** to stop the process you are running in the shell instead of binding Stop Main Process to this key which would stop the shell itself.

Shell (bound to **C-M-s**) [*Command*]
Shell Utility (initial value **`"/bin/csh"`**) [Hemlock *Variable*]
Shell Utility Switches (initial value **`nil`**) [Hemlock *Variable*]
Current Shell [Hemlock *Variable*]
Ask about Old Shells [Hemlock *Variable*]

This command executes the process determined by the values of Shell Utility and Shell Utility Switches in a new buffer named **"Shell n"** where **"n"** is some distinguishing integer.

Current Shell is a Hemlock variable that holds to the current shell buffer. When Shell is invoked, if there is a Current Shell, the command goes to that buffer.

When there is no Current Shell, but shell buffers do exist, if Ask about Old Shells is set, the Shell command prompts for one of them, setting Current Shell to the indicated shell, and goes to the buffer.

Invoking Shell with an argument forces the creation of a new shell buffer.

Shell Utility is the string name of the process to execute.

Shell Utility Switches is a string containing the default command line arguments to Shell Utility. This is a string since the utility is typically **"/bin/csh"**, and this string can contain I/O redirection and other shell directives.

Shell Command Line in Buffer [*Command*]

This command prompts for a buffer and a shell command line. It then runs a shell, giving it the command line, in the buffer.

Set Current Shell [*Command*]

This command sets the value of Current Shell.

Stop Main Process [*Command*]

This command stops the process running in the current buffer by sending a **:SIGTSTP** to that process. With an argument, stops the process using **:SIGSTOP**.

Continue Main Process [*Command*]

If the process in the current buffer is stopped, this command continues it.

Kill Main Process [*Command*]
Kill Process Confirm (initial value **t**) [Hemlock *Variable*]

This command prompts for confirmation and kills the process running in the current buffer.

Setting this variable to **nil** inhibits Hemlock's prompting for confirmation.

Stop Buffer Subprocess (bound to **H-z** in Process mode) [*Command*]

This command stops the foreground subprocess of the process in the current buffer, similar to the effect of **C-Z** in a shell.

Kill Buffer Subprocess [*Command*]

This command kills the foreground subprocess of the process in the current buffer.

Interrupt Buffer Subprocess (bound to **H-c** in Process mode) [*Command*]

This command interrupts the foreground subprocess of the process in the current buffer, similar to the effect of **C-C** in a shell.

Quit Buffer Subprocess (bound to **H-** in Process mode) [*Command*]

This command dumps the core of the foreground subprocess of the processs in the current buffer, similar to the effect of **C-** in a shell.

Send EOF to Process (bound to **H-d** in Process mode) [*Command*]

This command sends the end of file character to the process in the current buffer, similar to the effect of **C-D** in a shell.

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

This command sends the text the user has inserted at the end of a process buffer to the process in that buffer. Resulting output is inserted at the end of the process buffer.

The user may edit process input using commands that are shared with Typescript mode, see section 9.2.

## 6.4. Bufed Mode

Hemlock provides a mechanism for managing buffers as an itemized list. Bufed supports conveniently deleting several buffers at once, saving them, going to one, etc., all in a key stroke.

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

This command creates a list of buffers in a buffer supporting operations such as deletion, saving, and selection. If there already is a Bufed buffer, this just goes to it.

Bufed Help [*Command*]

This command pops up a display of Bufed help.

Bufed Delete (bound to **Bufed: C-d, C-D, D, d**) [*Command*]
Virtual Buffer Deletion (initial value `t`) [Hemlock *Variable*]
Bufed Delete Confirm (initial value `t`) [Hemlock *Variable*]

Bufed Delete deletes the buffer on the current line.

When Virtual Buffer Deletion is set, this merely flags the buffer for deletion until Bufed Expunge or Bufed Quit executes.

Whenever these commands actually delete a buffer, if Bufed Delete Confirm is set, then Hemlock prompts the user for permission; if more than one buffer is flagged for deletion, this only prompts once. For each modified buffer, Hemlock asks to save the buffer before deleting it.

Bufed Undelete (bound to **Bufed: U, u**) [*Command*]

This command undeletes the buffer on the current line.

Bufed Expunge (bound to **Bufed: !**) [*Command*]

This command expunges any buffers marked for deletion regarding Bufed Delete Confirm.

Bufed Quit (bound to **Bufed: q**) [*Command*]

This command kills the Bufed buffer, expunging any buffers marked for deletion.

Bufed Goto (bound to **Bufed: Space**) [*Command*]

This command selects the buffer on the current line, switching to it.

Bufed Goto and Quit (bound to **Bufed: S-leftdown**) [*Command*]

This command goes to the buffer under the pointer, quitting Bufed. It supplies a function for Generic Pointer Up which is a no-op.

Bufed Save File (bound to **Bufed: s**) [*Command*]

This command saves the buffer on the current line.

## 6.5. Completion

This is a minor mode that saves words greater than three characters in length, allowing later completion of those words. This is very useful for the often long identifiers used in Lisp programs. As you type a word, such as a Lisp symbol when in Lisp mode, and you progress to typing the third letter, Hemlock displays a possible completion in the status line. You can then rotate through the possible completions or type some more letters to narrow down the possibilities. If you choose a completion, you can also rotate through the possibilities in the buffer instead of in the status line. Choosing a completion or inserting a character that delimits words moves the word forward in the ring of possible completions, so the next time you enter its initial characters, Hemlock will prefer it over less recently used completions.

Completion Mode [*Command*]

This command toggles Completion mode in the current buffer.

Completion Self Insert [*Command*]

This command is like Self Insert, but it also checks for possible completions displaying any result in the status line. This is bound to most of the key-events with corresponding graphic characters.

Completion Complete Word (bound to **Completion: End**) [*Command*]

This command selects the currently displayed completion if there is one, guessing the case of the inserted text as with Query Replace. Invoking this immediately in succession rotates through possible completions in the buffer. If there is no currently displayed completion on a first invocation, this tries to find a completion from text immediately before the point and displays the completion if found.

Completion Rotate Completions (bound to **Completion: M-End**) [*Command*]

This command displays the next possible completion in the status line. If there is no currently displayed completion, this tries to find a completion from text immediately before the point and displays the completion if found.

List Possible Completions [*Command*]

This command lists all the possible completions for the text immediately before the point in a pop-up display. Sometimes this is more useful than rotating through several completions to see if what you want is available.

Completion Bucket Size (initial value **`20`**) [Hemlock *Variable*]

Completions are stored in buckets determined by the first three letters of a word. This variable limits the number of completions saved for each combination of the first three letters of a word. If you have many identifier in some module beginning with the same first three letters, you'll need increase this variable to accommodate all the names.

Save Completions [*Command*]
Read Completions [*Command*]
Completion Database Filename (initial value **nil**) [Hemlock *Variable*]

Save Completions writes the current completions to the file Completion Database Filename. It writes them, so Read Completions can read them back in preserving the most-recently-used order. If the user supplies an argument, then this prompts for a pathname.

Read Completions reads completions saved in Completion Database Filename. It moves any current completions to a less-recently-used status, and it removes any in a given bucket that exceed the limit Completion Bucket Size.

Parse Buffer for Completions [*Command*]

This command passes over the current buffer putting each valid completion word into the database. This is a good way of picking up many useful completions upon visiting a new file for which there are no saved completions.

## 6.6. CAPS-LOCK Mode

CAPS-LOCK is a minor mode in which Hemlock that inserts all alphabetic characters as uppercase letters.

Caps Lock Mode [*Command*]

This command toggles CAPS-LOCK mode for the current buffer; it is most useful when bound to a key, so you can enter and leave CAPS-LOCK mode casually.

Self Insert Caps Lock [*Command*]

This command inserts the uppercase version of the character corresponding to the last key-event typed.

## 6.7. Overwrite Mode

Overwrite mode is a minor mode which is useful for creating figures and tables out of text. In this mode, typing a key-event with a corresponding graphic character replaces the character at the point instead of inserting the character. Quoted Insert 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 [*Command*]

This command replaces the next character with the character corresponding to the key-event used to invoke the command. After replacing the character, this moves past it. If the next character is a tab, this first expands the tab into the appropriate number of spaces, replacing just the next space character. At the end of the line, it inserts the character instead of clobbering the newline.

This is bound to key-events with corresponding graphic characters in Overwrite mode.

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. This deletes tabs and newlines.

## 6.8. 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.

### 6.8.1. 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.

### 6.8.2. 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.

Abbrev Pathname Defaults (initial value **`(pathname "abbrev.defns")`**) [Hemlock *Variable*]

This is sticky default for the following commands. When they prompt for a file to write, they offer this and set it for the next time one of them executes.

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.

### 6.8.3. 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.

## 6.8.4. 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.

## 6.8.5. Deleting Word Abbrevs

The user may delete word abbrevs 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 abbrev definitions.

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.

## 6.9. Lisp Library

This is an implementation dependent feature. The Lisp library is a collection of local hacks that users can submit and share that is maintained by the Lisp group. These commands help peruse the catalog or description files and figure out how to load the entries.

Lisp Library [*Command*]

This command finds all the library entries and lists them in a buffer. The following commands describe and load those entries.

Describe Library Entry (bound to **Lisp-Lib: space**) [*Command*]
Describe Pointer Library Entry (bound to **Lisp-Lib: leftdown**) [*Command*]
Load Library Entry (bound to **Lisp-Lib: rightdown**) [*Command*]
Load Pointer Library Entry (bound to **Lisp-Lib: l**) [*Command*]
Editor Load Library Entry [*Command*]
Editor Load Pointer Library Entry [*Command*]

Load Library Entry and Load Pointer Library Entry load the library entry indicated by the line on which the point lies or where the user clicked the pointer, respectively. These load the entry into the current slave Lisp.

Editor Load Library Entry and Editor Load Pointer Library Entry are the same, but they load the entry into the editor Lisp.

Exit Lisp Library (bound to **Lisp-Lib: q**) [*Command*]

This command deletes the Lisp Library buffer.

Lisp Library Help (bound to **Lisp-Lib: ?**) [*Command*]

This command pops up a help window listing Lisp-Lib commands.

# 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 this moves the point to the start of the comment. If there no comment, this creates an empty one.

This command normally indents the comment to start at Comment Column. The comment indents differently in the following 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 remains unmoved.
2. If the last character in the Comment Start appears two times, then the comment is indented like a line of code.
3. If text on the line prevents the comment occurring in the desired position, this places the comment 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, you can exploit these properties 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 or leading whitespace at the beginning of lines. The Hemlock indentation facility is integrated into the command set so that it 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).

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.

Indent Region (bound to **C-M-**\) [*Command*]

This command indents every line in the region. It may be undone with Undo.

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 this leaves one space between the two joined lines, 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 this inserts no space.
2. If the preceding character is a newline, then this inserts no space. This will happen if the

previous line was blank.

3. If the preceding character is a sentence terminator, then this inserts two spaces.

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

Quote Tab (bound to **M-Tab**) [*Command*]

This command inserts a tab character.

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.

Center Line [*Command*]

This indents the current line so that it is centered between the left margin and Fill Column (page 38). If a prefix argument is supplied, then it is used as the width instead of Fill Column.

Indent Function (initial value **`tab-to-tab-stop`**) [Hemlock *Variable*]

The value of this variable determines how indentation is done, and it 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`**. The default simply inserts a tab character at the mark.

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. The function will insert a maximum number of tabs and a minimum number of spaces at mark to move the specified number of columns. The default definition uses Spaces per Tab to determine the size of a tab. *Note,* Spaces per Tab *is not used everywhere in* Hemlock *yet, so changing this variable could have unexpected results.*

## 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. Whenever the user or some Hemlock mechanism turns on Lisp mode, the mode's setup includes locally setting Current Package (see section 9.3) in that buffer if its value is non-existent there; the value used is **"USER"**.

Lisp Mode [*Command*]

This command sets 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.

Extract Form [*Command*]

This command replaces the current containing list with the next form. The entire affected area is pushed onto the kill ring. If an argument is supplied, that many upward levels of list nesting is replaced by the next form. This is similar to Extract List, but this command is more generally useful since it works on any kind of form; it is also more intuitive since it operates on the next form as many Lisp mode commands do.

## 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 hard to manage, and inconsistently indented Lisp is subtly misleading. See section 7.2 for a description of the general-purpose indentation commands. Lisp mode uses these indentation rules:

- If in a semicolon (**;**) comment, then use the standard comment indentation rules. See page 61.
- If in a quoted string, then indent to the column one greater than the column containing the opening double quote. This is exactly what you want in function documentation strings and wrapping **error** strings.
- 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 from 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. This is undo-able.

Fill Lisp Comment Paragraph (bound to **M-q** in Lisp mode) [*Command*]
Fill Lisp Comment Paragraph Confirm (initial value **t**) [Hemlock *Variable*]

This fills a flushleft or indented Lisp comment. This also fills Lisp string literals using the proper indentation as a filling prefix. When invoked outside of a comment or string, this tries to fill all contiguous lines beginning with the same initial, non-empty blankspace. When filling a comment, the current line is used to determine a fill prefix by taking all the initial whitespace on the line, the semicolons, and any whitespace following the semicolons.

When invoked outside of a comment or string, this command prompts for confirmation before filling. It is useful to use this for filling long **`export`** lists or other indented text or symbols, but since this is a less common use, this command tries to make sure that is what you wanted. Setting Fill Lisp Comment Paragraph Confirm to **`nil`** inhibits the confirmation prompt.

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*. Two different styles of parenthesis matching are supported: highlighting and pausing.

Highlight Open Parens (initial value **`t`**) [Hemlock *Variable*]
Open Paren Highlighting Font (initial value **`nil`**) [Hemlock *Variable*]

When Highlight Open Parens is true, and a close paren is immediately before the point, then Hemlock displays the matching open paren in Open Paren Highlighting Font.

Open Paren Highlighting Font is the string name of the font used for paren highlighting. Only the "**`(`**" character is used in this font. If null, then a reasonable default is chosen. The highlighting font is read at initialization time, so this variable must be set before the editor is first entered to have any effect.

Lisp Insert ) (bound to **`)`** in Lisp mode) [*Command*]
Paren Pause Period (initial value **`0.5`**) [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. Paren pausing may be disabled by setting Paren Pause Period to **`nil`**.

The initial values shown for Highlight Open Parens and Paren Pause Period are only approximately correct. Since paren highlighting is only meaningful in Lisp mode, Highlight Open Parens is false globally, and has a mode-local value of **`t`** in Lisp mode. It it redundant to do both kinds of paren matching, so there is also a binding of Paren Pause Period to **`nil`** in Lisp mode.

Paren highlighting is only supported under X windows, so the above defaults are conditional on the device type. If Hemlock is started on a terminal, the initialization code makes Lisp mode bindings of **`nil`** and **`0.5`** for Highlight Open Parens and Paren Pause Period. Since these alternate default bindings are made at initialization time, the only way to affect them is to use the **`after-editor-initializations`** macro.

## 8.7. Parsing Lisp

Lisp mode has a 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, while 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 **`500`**) [Hemlock *Variable*]
Minimum Lines Parsed (initial value **`50`**) [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 commands usually act 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 Unix TCP sockets 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 debugging a Lisp program.
- Editing may occur while running a Lisp program.
- The eval server may be on a different machine, removing the load from the editing machine.
- Multiple eval servers allow the use of several distinct Lisp environments.

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

### 9.1.1. 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. 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. The current eval server is normally globally specified, but it may also be shadowed locally in specific buffers.

Set Eval Server [*Command*]
Set Buffer Eval Server [*Command*]
Current 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. Current Eval Server displays the name of the current eval server in the echo area, taking any buffer eval server into consideration. See also Set Compile Server (page 77).

### 9.1.2. Slaves

For now, all eval servers are *slaves*. A slave is a Lisp process that uses a typescript (see page 73) to run its top-level **`read-eval-print`** loop in a Hemlock buffer. We refer to the buffer that a slave uses for I/O as its *interactive* or *slave* buffer. The name of the interactive buffer is the same as the eval server's name.

Hemlock creates a *background* buffer for each eval server. 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 interactive buffer.

Hemlock locally sets Current Eval Server in interactive and background buffers to their associated slave. When in a slave or background buffer, eval server requests will go to the associated slave, regardless of the global value of Current Eval Server.

Select Slave (bound to **C-M-c**) [*Command*]

This command changes the current buffer to the current eval server's interactive buffer. If the current eval server is not a slave, then it beeps. If there is no current eval server, then this creates a slave (see section 9.1.3). If a prefix argument is supplied, then this creates a new slave regardless of whether there is a current eval server. This command is the standard way to create a slave.

The slave buffer is a typescript (see page 73) the slave uses for its 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 it beeps.

### 9.1.3. Slave Creation and Destruction

When Hemlock first starts up, there is no current eval server. If there is no a current eval server, commands that need to use the current eval server will create a slave as the current eval server.

If an eval server's Lisp process terminates, then we say the eval server is dead. Hemlock displays a message in the echo area, interactive, and background buffers whenever an eval server dies. If the user deletes an interactive or background buffer, the associated eval server effectively becomes impotent, but Hemlock does not try to kill the process. If a command attempts to use a dead eval server, then the command will beep and display a message.

Confirm Slave Creation (initial value **`t`**) [Hemlock *Variable*]

If this variable is true, then Hemlock always prompts the user for confirmation before creating a slave.

Ask About Old Servers (initial value **`t`**) [Hemlock *Variable*]

If this variable is true, and some slave already exists, Hemlock prompts the user for the name of an existing server when there is no current server, instead of creating a new one.

Editor Server Name [*Command*]

This command echos the editor server's name, the machine and port of the editor, which is suitable for use with the Lisp processes -slave switch. See section 9.10.

Accept Slave Connections [*Command*]

This command cause Hemlock to accept slave connections, and it displays the editor server's name, which is suitable for use with the Lisp processes -slave switch. See section 9.10. Supplying an argument causes this command to inhibit slave connections.

Slave Utility (initial value **"/usr/misc/.lisp/bin/lisp"**) [Hemlock *Variable*]
Slave Utility Switches [Hemlock *Variable*]

A slave is started by running the program Slave Utility Name with arguments specified by the list of strings Slave Utility Switches. This is useful primarily when running customized Lisp systems. For example, setting Slave Utility Switches to **("-core" "my.core")** will cause "**/usr/hqb/my.core**" to be used instead of the default core image.

The **-slave** switch and the editor name are always supplied as arguments, and should remain unspecified in Slave Utility Switches.

Kill Slave [*Command*]
Kill Slave and Buffers [*Command*]

Kill Slave prompts for a slave name, aborts any operations in the slave, tells the slave to **quit**, and shuts down the connection to the specified eval server. This makes no attempt to assure the eval server actually dies.

Kill Slave and Buffers is the same as Kill Slave, but it also deletes the interactive and background buffers.

### 9.1.4. 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 interaction in a Hemlock buffer similar to that of a terminal. When there is a typescript in a buffer, the Typescript minor mode is present. Some of the commands described in this section are also used by Eval mode (page 81.)

Typescripts are simple to use. Hemlock inserts output from the process into the buffer. To give the process input, use normal editing to insert the input at the end of the buffer, and then type **Return** to confirm sending the input to the process.

Confirm Typescript Input (bound to **Return** in Typescript mode) [*Command*]
Unwedge Interactive Input Confirm (initial value **t**) [Hemlock *Variable*]

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, Hemlock moves the point to the end of the buffer and inserts a newline.

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 the process reading on the buffer's typescript is not waiting 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.

If the buffer's point is before the start of the input area, then various actions can occur. When set, Unwedge Interactive Input Confirm causes Hemlock to ask the user if it should fix the input buffer which typically results in ignoring any current input and refreshing the input area at the end of the buffer. This also has the effect of throwing the slave Lisp to top level, which aborts any pending operations or queued input. This is the only way to be sure the user is cleanly set up again after messing up the input region. When this is **nil**, Hemlock simply beeps and tells the user in the Echo Area that the input area is invalid.

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**.

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*]
Search Previous Interactive Input (bound to **M-P** in Typescript and Eval modes) [*Command*]
Interactive History Length (initial value **10**) [Hemlock *Variable*]
Minimum Interactive Input Length (initial value **2**) [Hemlock *Variable*]

Hemlock maintains a history of interactive inputs. Next Interactive Input and Previous Interactive Input step forward and backward in the history, inserting the current entry in the buffer. The prefix argument is used as a repeat count.

Search Previous Interactive Input searches backward through the interactive history using the current input as a search string. Consecutive invocations repeat the previous search.

Interactive History Length determines the number of entries with which Hemlock creates the buffer-specific histories. Hemlock only adds an input region to the history if its number of characters exceeds Minimum Interactive Input Length.

Reenter Interactive Input (bound to **C-Return** in Typescript and Eval modes) [*Command*]

This copies to the end of the buffer the form to the left of the buffer's point. When the current region is active, this copies it instead. This is sometimes easier to use to get a previous input that is either so far back that it has fallen off the history or is visible and more readily *yanked* than gotten with successive invocations of the history commands.

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*]
Slave GC Alarm (initial value **:message**) [Hemlock *Variable*]

Input Wait Alarm determines what action to take when a slave Lisp goes into an input wait on a typescript that isn't currently displayed in any window. Slave GC Alarm determines what action to take when a slave notifies that it is GC'ing.

The following are legal values:

**: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.

Typescript Slave BREAK (bound to **Typescript: H-b**) [*Command*]
Typescript Slave to Top Level (bound to **Typescript: H-g**) [*Command*]
Typescript Slave Status (bound to **Typescript: H-s**) [*Command*]

Some typescripts have associated information which these commands access allowing Hemlock to control the process which uses the typescript.

Typescript Slave BREAK puts the current process in a break loop so that you can be debug it. This is similar in effect to an interrupt signal (**^C** or **^** in the editor process).

Typescript Slave to Top Level causes the current process to throw to the top-level **read-eval-print** loop. This is similar in effect to a quit signal (**^**).

Typescript Slave Status causes the current process to print status information on ***error-output***:

```
; Used 0:06:03, 3851 faults.  In: SYSTEM:SERVE-EVENT
```

The message displays the process run-time, the total number of page faults and the name of the currently running function. This command is useful for determining whether the slave is in an infinite loop, waiting for input, or whatever.

## 9.3. The Current Package

The current package is the package which Lisp interaction commands use. The current package is specified on a per-buffer basis, and defaults to "**USER**". 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. The current package is displayed in the modeline (see section 1.6.4.) Normally the package for each file is specified using the **Package** file option (see page 34.)

When in a slave buffer, the current package is controlled by the value of ***package*** in that Lisp process. Modeline display of the current package is inhibited in this case.

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, background, or eval buffer, then this sets the current package in the associated eval server or editor Lisp. When in an interactive buffer, do not use **in-package**; use this command instead.

## 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. If the region is active, then Evaluate Defun evaluates the current region, just like Evaluate Region.

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

This command shows the macroexpansion of the next expression in the null environment in a pop-up window. With an argument, it uses **`macroexpand`** instead of **`macroexpand-1`**.

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. If the region is active, then Compile Defun compiles the current region, just like Compile Region.

Load File [*Command*]
Load Pathname Defaults (initial value **`nil`**) [Hemlock *Variable*]

This command prompts for a file and loads it into the current eval server using **`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. This command also uses the Remote Compile File variable.

## 9.5. Compiling Files

These commands are used to compile source ("**`.lisp`**") files, producing binary ("**`.fasl`**") output files. Note that unlike the other compiling and evalating commands, this does not have the effect of placing the definitions in the environment; to do so, the binary file must be loaded.

Compile Buffer File (bound to **C-x c**) [*Command*]
Compile Buffer File Confirm (initial value **t**) [Hemlock *Variable*]

This command asks for confirmation, then saves the current buffer (when modified) and compiles the associated file. The confirmation prompt indicates intent to save and compile or just compile. If the buffer wasn't modified, and a comparison of the write dates for the source and corresponding binary ("**.fasl**") file suggests that recompilation is unnecessary, the confirmation also indicates this. A prefix argument overrides this test and forces recompilation. Since there is a complete log of output in the background buffer, the creation of the normal error output ("**.err**") file is inhibited.

Setting Compile Buffer File Confirm to **nil** inhibits confirmation, except when the binary is up to date and a prefix argument is not supplied.

Compile File [*Command*]

This command prompts for a file and compiles that file, providing a convenient way to compile a file that isn't in any buffer. Unlike Compile Buffer File, this command doesn't do any consistency checks such as checking whether the source is in a modified buffer or the binary is up to date.

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*]
Current Compile Server [*Command*]

These commands are analogous to Set Eval Server, Set Buffer Eval Server (page 71) and Current Eval Server, but they determine the eval server used for file compilation requests. If the user specifies a compile server, then the file compilation commands 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 while continuing to interact with your eval server and editor. The compile server can also run on a remote machine relieving your active development machine of the compilation effort.

Next Compiler Error (bound to **H-n**) [*Command*]
Previous Compiler Error (bound to **H-p**) [*Command*]

These commands provides a convenient way to inspect compiler errors. First it splits the current window if there is only one window present. Hemlock positions the current point in the first window at the erroneous source code for the next (or previous) error. Then in the second window, it displays the error beginning at the top of the window. Given an argument, this command skips that many errors.

Flush Compiler Error Information [*Command*]

This command relieves the current eval server of all infomation about errors encountered while compiling. This is convenient if you have been compiling a lot, but you were ignoring errors and warnings. You don't want to step through all the old errors, so you can use this command immediately before compiling a file whose errors you intend to edit.

Remote Compile File (initial value **`nil`**) [Hemlock *Variable*]

When true, this variable causes file compilations to be done using the RFS remote file system mechanism by prepending "`/../`*host*" to the file being compiled. This allows the compile server to be run on a different machine, but requires that the source be world readable. If false, commands use source filenames directly. Do NOT use this to compile files in AFS.

## 9.6. 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. If there is no currently valid eval server, then this command uses the editor Lisp's environment instead of trying to spawn a slave.

## 9.7. 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*]
Goto Definition (bound to **C-M-F**) [*Command*]
Edit Command Definition [*Command*]

Edit Definition prompts for the name of a function, and then uses the current eval server to find out in which file the function is defined. If something other than **`defun`** or **`defmacro`** defined the function, then this simply reads in the file, without trying to find its definition point within the file. If the function is uncompiled, then this looks for it in the current buffer. If there is no currently valid eval server, then this command uses the editor Lisp's environment instead of trying to spawn a slave.

Goto Definition edits the definition of the symbol at the beginning of the current list.

Edit Command Definition edits the definition of a Hemlock command. By default, this command does a keyword prompt for the command name (as in an extended command). If a prefix argument is specified, then instead prompt for a key and edit the definition of the command bound to that key.

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.

Editor Definition Info (initial value **nil**) [Hemlock *Variable*]

When this variable is true, the editor Lisp is used to determine definition editing information, otherwise the current eval server is used. This variable is true in Eval and Editor modes.

## 9.8. Debugging

These commands manipulate the slave when it is in the debugger and provide source editing based on the debugger's current frame. These all affect the Current Eval Server.

### 9.8.1. Changing Frames

Debug Down (bound to **C-M-H-d**) [*Command*]

This command moves down one debugger frame.

Debug Up (bound to **C-M-H-u**) [*Command*]

This command moves up one debugger frame.

Debug Top (bound to **C-M-H-t**) [*Command*]

This command moves to the top of the debugging stack.

Debug Bottom (bound to **C-M-H-b**) [*Command*]

This command moves to the bottom of the debugging stack.

Debug Frame (bound to **C-M-H-f**) [*Command*]

This command moves to the absolute debugger frame number indicated by the prefix argument.

### 9.8.2. Getting out of the Debugger

Debug Quit (bound to **C-M-H-q**) [*Command*]

This command throws to top level out of the debugger in the Current Eval Server.

Debug Go (bound to **C-M-H-g**) [*Command*]

This command tries the **continue** restart in the Current Eval Server.

Debug Abort (bound to **C-M-H-a**) [*Command*]

This command executes the ABORT restart in the Current Eval Server.

Debug Restart (bound to **C-M-H-r**) [*Command*]

This command executes the restart indicated by the prefix argument in the Current Eval Server. The debugger enumerates the restart cases upon entering it.

### 9.8.3. Getting Information

Debug Help (bound to **C-M-H-h**) [*Command*]

This command in prints the debugger's help text.

Debug Error (bound to **C-M-H-e**) [*Command*]

This command prints the error condition and restart cases displayed upon entering the debugger.

Debug Backtrace (bound to **C-M-H-B**) [*Command*]

This command executes the debugger's **`backtrace`** command.

Debug Print (bound to **C-M-H-p**) [*Command*]

This command prints the debugger's current frame in the same fashion as the frame motion commands.

Debug Verbose Print (bound to **C-M-H-P**) [*Command*]

This command prints the debugger's current frame without elipsis.

Debug Source (bound to **C-M-H-s**) [*Command*]

This command prints the source form for the debugger's current frame.

Debug Verbose Source [*Command*]

This command prints the source form for the debugger's current frame with surrounding forms for context.

Debug List Locals (bound to **C-M-H-l**) [*Command*]

This prints the local variables for the debugger's current frame.

### 9.8.4. Editing Sources

Debug Edit Source (bound to **C-M-H-S**) [*Command*]

This command attempts to place you at the source location of the debugger's current frame. Not all debugger frames represent function's that were compiled with the appropriate debug-info policy. This beeps with a message if it is unsuccessful.

### 9.8.5. Miscellaneous

Debug Flush Errors (bound to **C-M-H-F**) [*Command*]

This command toggles whether the debugger ignores errors or recursively enters itself.

## 9.9. 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.

Editor Load File [*Command*]

This command is analogous to Load File (page 76), but loads the file into the editor process.

### 9.9.1. 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 **Mode** file option (see page 34.)

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 Macroexpand Expression (bound to **Editor: C-M**) [*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 Buffer File [*Command*]
Editor Compile File [*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.

### 9.9.2. 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 71), 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 binds **`*standard-output*`** to a stream that inserts into the buffer and **`*standard-input*`** to a stream that signals an error for all operations. Hemlock cannot correctly support the interactive evaluation of forms that read from the Eval interactive buffer.

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.

Confirm 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. This beeps if the form is incomplete. Use **Linefeed** to insert line breaks in the middle of a form.

This command uses Unwedge Interactive Input Confirm in the same way Confirm Interactive Input does.

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.

### 9.9.3. 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.

**e** Display the original error message in a pop-up window.

**b** Show a stack backtrace in a pop-up window.

**q, Escape** Quit from this error to the nearest command loop.

**r** Display a list of the restart cases and prompt for the number of a **`restart-case`** with which to continue. Restarting may result in prompting in the window in which Lisp started.

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 73). Errors due to interaction in a slave buffer will cause the debugger to be entered in the slave buffer.

## 9.10. Command Line Switches

Two command line switches control the initialization of editor and eval servers for a Lisp process:

**`-edit`** This switch starts up Hemlock. If there is a non-switch command line word immediately following the program name, then the system interprets it as a file to edit. For example, given

```
lisp file.txt -edit
```

Lisp will go immediately into Hemlock finding the file **`file.txt`**.

**`-slave [`***name***`]`** This switch causes the Lisp process to become a slave of the editor process *name*. An editor Lisp determines *name* when it allows connections from slaves. Once the editor chooses a name, it keeps the same name until the editor's Lisp process terminates. 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. **`hqb`**'s machine is **`ME.CS.CMU.EDU`**, and he wants want to run a slave on **`SLAVE.CS.CMU.EDU`**, then he should use the Accept Slave Connections command, telnet to the machine, and invoke Lisp supplying **`-slave`** and the editor's name. The command displays the editor's name.

# Chapter 10

# The Mail Interface

## 10.1. Introduction to Mail in Hemlock

Hemlock provides an electronic mail handling facility via an interface to the public domain *Rand MH Message Handling System*. This chapter assumes that the user is familiar with the basic features and operation of MH, but it attempts to make allowances for beginners. Later sections of this chapter discuss setting up MH, profile components and special files for formatting outgoing mail headers, and backing up protected mail directories on a workstation. For more information on MH, see the *Rand MH Message Handling System Tutorial* and the *Rand MH Message Handling System Manual*.

The Hemlock interface to MH provides a means for generating header (**scan**) lines for messages and displaying these headers in a Headers buffer. This allows the user to operate on the *current message* as indicated by the position of the cursor in the Headers buffer. The user can read, reply to, forward, refile, or perform various other operations on the current message. A user typically generates a Headers buffer with the commands Message Headers or Incorporate and Read New Mail, and multiple such buffers may exist simultaneously.

Reading a message places its text in a Message buffer. In a manner similar to a Headers buffer, this allows the user to operate on that message. Most Headers buffer commands behave the same in a Message buffer. For example, the Reply to Message command has the same effect in both Headers mode and Message mode. It creates a Draft buffer and makes it the current buffer so that the user may type a reply to the current message.

The Send Message command originates outgoing mail. It generates a Draft buffer in which the user composes a mail message. Each Draft buffer has an associated pathname, so the user can save the buffer to a file as necessary. Invoking Send Message in a Headers or Message buffer associates the Draft buffer with a Message buffer. This allows the user to easily refer to the message being replied to with the command Goto Message Buffer. After the user composes a draft message, he can deliver the message by invoking the Deliver Message command in the Draft buffer (which deletes both the this buffer and any associated Message buffer), or he can delay this action. Invoking Deliver Message when not in a Draft buffer causes it to prompt for a draft message ID, allowing previously composed and saved messages to be delivered (even across distinct Lisp invocations).

The Hemlock mail system provides a mechanism for *virtual message deletion*. That is, the Delete Message command does not immediately delete a message but merely flags the message for future deletion. This allows the user to undelete the messages with the Undelete Message command. The Expunge Messages command actually removes messages flagged for deletion. After expunging a deleted message, Undelete Messages can no longer retrieve it. Commands that read messages by sequencing through a Headers buffer typically ignore those marked for deletion, which makes for more fluid reading if a first pass has been made to delete uninteresting messages.

After handling messages in a Headers buffer, there may be messages flagged for deletion and possibly multiple

Message buffers lying around. There is a variety of commands that help *terminate* a mail session. Expunge Messages will flush the messages to be deleted, leaving the buffer in an updated state. Delete Headers Buffer and Message Buffers will delete the Headers buffer and its corresponding Message buffers. Quit Headers is a combination of these two commands in that it first expunges messages and then deletes all the appropriate buffers.

One does not have to operate only on messages represented in a Headers buffer. This is merely the nominal mode of interaction. There are commands that prompt for a folder, an MH message specification (for example, "**`1 3 6 last`**", "**`1-3 5 6`**", "**`all`**", "**`unseen`**"), and possibly a **`pick`** expression. **`Pick`** expressions allow messages to be selected based on header field pattern matching, body text searching, and date comparisons; these can be specified using either a Unix shell-like/switch notation or a Lisp syntax, according to one's preference. See section 10.7 for more details.

A *mail-drop* is a file where a Unix-based mail system stores all messages a user receives. The user's mail handling program then fetches these from the mail-drop, allowing the user to operate on them. Traditionally one locates his mail-drop and mail directory on a mainframe machine because the information on mainframes is backed up on magnetic tape at least once per day. Since Hemlock only runs under CMU COMMON LISP on workstations, and one's mail directory is not usually world writable, it is not possible to adhere to a standard arrangement. Since MH provides for a remote mail-drop, and CMU's Remote File System has a feature allowing authentication across a local area network, one can use Hemlock to fetch his mail from a mainframe mail-drop (where it is backed up before Hemlock grabs it) and store it on his workstation. Reading mail on a workstation is often much faster and more comfortable because typically it is a single user machine. Section 10.5 describes how to back up one's mail directory from a workstation to a mainframe.

## 10.2. Constraints on MH to use Hemlock's Interface

There are a couple constaints placed on the user of the Hemlock interface to MH. The first is that there must be a draft folder specified in one's MH profile to use any command that sends mail. Also, to read new mail, there must be an **`Unseen-Sequence:`** component in one's MH profile. The default MH profile does not specify these components, so they must be added by the user. The next section of this chapter describes how to add these components. Another constraint is that Hemlock requires its own **`scan`** line format to display headers lines in a Headers buffer. See the description of the variable MH Scan Line Form for details.

## 10.3. Setting up MH

Get an MH default profile and mail directory by executing the MH **`folder`** utility in a Unix shell. When it asks if it should make the "**`inbox`**" folder, answer "**yes**". This creates a file called "**`.mh_profile`**" in the user's home directory and a directory named "**`Mail`**".

Edit the "**`.mh_profile`**" file inserting two additional lines. To send mail in Hemlock, the user must indicate a draft folder by adding a **`Draft-Folder:`** line with a draft folder name — "**`drafts`**" is a common name:

```
Draft-Folder: drafts
```

Since the mail-drop exists on a remote machine, the following line must also be added:

```
MailDrop: /../<hostname>/usr/spool/mail/<username>
```

Since the user's mail-drop is on a separate machine from his mail directory (and where the user runs Hemlock), it is necessary to issue the following command from the Unix shell (on the workstation). This only needs to be done once.

```
/usr/cs/etc/rfslink -host <hostname> /usr/spool/mail/<username>
```

Note that **<hostname>** is not a full ARPANET domain-style name. Use an abbreviated CMU host name (for example, "**spice**" not "**spice.cs.cmu.edu**").

## 10.4. Profile Components and Customized Files

### 10.4.1. Profile Components

The following are short descriptions about profile components that are either necessary to using Hemlock's interface to MH or convenient for using MH in general:

**`Path:`** This specifies the user's mail directory. It can be either a full pathname or a pathname relative to the user's home directory. This component is *necessary* for using MH.

**`MailDrop:`** This is used to specify one's remote mail-drop. It is *necessary* for Hemlock only when using a mail-drop other than "**`/usr/spool/mail/<user>`**" on the local machine.

**`Folder-Protect:`**, **`Msg-Protect:`**
These are set to 700 and 600 respectively to keep others from reading one's mail. At one time the default values were set for public visibility of mail folders. Though this is no longer true, these can be set for certainty. The 700 protection allows only user read, write, and execute (list access for directories), and 600 allows only user read and write. These are not necessary for either MH or the Hemlock interface.

**`Unseen-Sequence:`**
When mail is incorporated, new messages are added to this sequence, and as these messages are read they are removed from it. This allows the user at any time to invoke an MH program on all the unseen messges of a folder easily. An example definition is:

```
Unseen-Sequence: unseen
```

Specifying an unseen-sequence is *necessary* to use Hemlock's interface to MH.

**`Alternate-Mailboxes:`**
This is not necessary for either MH or the Hemlock interface. This component tells MH which addresses that it should recognize as the user. This is used for **`scan`** output formatting when the mail was sent by the user. It is also used by **`repl`** when it sets up headers to know who the user is for inclusion or exclusion from **cc**: lists. This is case sensitive and takes wildcards. One example is:

```
Alternate-Mailboxes: *FRED*, *Fred*, *fred*
```

**`Draft-Folder:`** This makes multiple draft creation possible and trivial to use. Just supply a folder name (for example, "**`drafts`**"). Specifying a draft-folder is *necessary* to use Hemlock's interface to MH.

**`repl: -cc all -nocc me -fcc out-copy`**
This tells the **`repl`** utility to include everyone but the user in the **cc:** list when replying to mail. It also makes **`repl`** keep an copy of the message the user sends. This is mentioned because one probably wants to reply to everyone receiving a piece of mail except oneself. Unlike other utilities that send mail, **`repl`** stores personal copies of outgoing mail based on a command line switch. Other MH utilities use different mechanisms. This line is not necessary to use either MH or the Hemlock interface.

**`rmmproc: /usr/cs/bin/rm`**
This is not necessary to use Hemlock's interface to MH, but due to Hemlock's virtual message deletion feature, this causes messages to be deleted from folder directories in a cleaner fashion when they actually get removed. Note that setting this makes **`rmm`** more treacherous if used in the Unix shell.

### 10.4.2. Components Files

*Components* files are templates for outgoing mail header fields that specify position and sometimes values for specified fields. Example files are shown for each one discussed here. These should exist in the user's mail directory.

For originating mail there is a components file named "**`components`**", and it is used by the MH utility **`comp`**. An example follows:

```
To:
cc:
fcc: out-copy
Subject:
--------
```

This example file differs from the default by including the **`fcc:`** line. This causes MH to keep a copy of the outgoing draft message. Also, though it isn't visible here, the **`To:`**, **`cc:`**, and **`Subject:`** lines have a space at the end.

The "**`forwcomps`**" components file is a template for the header fields of any forwarded message. Though it may be different, our example is the same as the previous one. These are distinct files for MH's purposes, and it is more flexible since the user might not want to keep copies of forwarded messages.

The "**`replcomps`**" components file is a template for the header fields of any draft message composed when replying to a message. An example follows:

```
%(lit)%(formataddr %<{reply-to}%|%<{from}%|%{sender}%>%>)\
%<(nonnull)%(void(width))%(putaddr To: )\n%>\
%(lit)%(formataddr{to})%(formataddr{cc})%(formataddr(me))\
%(formataddr{resent-to})\
%<(nonnull)%(void(width))%(putaddr cc: )\n%>\
%<{fcc}Fcc: %{fcc}\n%>\
%<{subject}Subject: Re: %{subject}\n%>\
%<{date}In-reply-to: Your message of \
%<(nodate{date})%{date}%|%(tws{date})%>.%<{message-id}
          %{message-id}%>\n%>\
--------
```

This example file differs from the default by including the **resent-to:** field (in addition to the **to:** and **cc:** fields) of the message being replied to in the **cc:** field of the draft. This is necessary for replying to all recipients of a distributed message. Keeping a copy of the outgoing draft message works a little differently with reply components. MH expects a switch which the user can put in his profile (see section 10.4.1 of this chapter), and using the MH formatting language, this file tests for the **`fcc`** value as does the standard file.

## 10.5. Backing up the Mail Directory

The easiest method of backing up a protected mail directory is to copy it into an Andrew File System (AFS) directory since these are backed up daily as with mainframes. The only problem with this is that the file servers may be down when one wants to copy his mail directory since, at the time of this writing, these servers are still under active development; however, they are becoming more robust daily. One can read about the current AFS status in the file **`/../fac/usr/gripe/doc/vice/status`**.

Using AFS, one could keep his actual mail directory (not a copy thereof) in his AFS home directory which eliminates the issue of backing it up. This is additionally beneficial if the user does not use the same workstation everyday (that is, he does not have his own but shares project owned machines). Two problems with this arrangement result from the AFS being a distributed file system. Besides the chance that the server will be down when the user wants to read mail, performance degrades since messages must always be referenced across the local area network.

Facilities' official mechanism for backing up protected directories is called **`sup`**. This is awkward to use and hard to set up, but a subsection here describes a particular arrangement suitable for the user's mail directory.

### 10.5.1. Andrew File System

If the user choses to use AFS, he should get copies of *Getting Started with the Andrew File System* and *Protecting AFS files and directories*. To use AFS, send mail to Gripe requesting an account. When Gripe replies with a password, change it to be the same as the account's password on the workstation. This causes the user to be authenticated into AFS when he logs into his workstation (that is, he is automatically logged into his AFS account). To change the password, first log into the AFS account:

```
log <AFS userid>
```

Then issue the **`vpasswd`** command.

All of the example command lines in this section assume the user has **`/usr/misc/bin`** on his Unix shell **`PATH`** environment variable.

**Copy into AFS:** Make an AFS directory to copy into:

```
mkdir /afs/cs.cmu.edu/user/<AFS userid>/mail-backup
```

This will be readable by everyone, so protect it with the following:

```
fs sa /afs/cs.cmu.edu/user/<AFSuserid>/mail-backup System:AnyUser none
```

Once the AFS account and directory to backup into have been established, the user needs a means to recursively copy his mail directory updating only those file that have changed and deleting those that no longer exist. To do this, issue the following command:

```
copy -2 -v -R <mail directory> <AFS backup directory>
```

Do not terminate either of these directory specifications with a **`/`**. The **`-v`** switch causes **`copy`** to output a line for copy and deletion, so this may be eliminated if the user desires.

**Mail Directory Lives in AFS:** Assuming the AFS account has been established, and the user has followed the directions in 10.3, now make an AFS directory to serve as the mail directory:

```
mkdir /afs/cs.cmu.edu/user/<AFS userid>/Mail
```

This will be readable by everyone, so protect it with the following:

```
fs sa /afs/cs.cmu.edu/user/<AFSuserid>/Mail System:AnyUser none
```

Tell MH where the mail directory is by modifying the profile's "**`.mh_profile`**" (see section 10.3) **`Path:`** component (see section 10.4.1):

```
Path: /afs/cs.cmu.edu/user/<AFS userid>/Mail
```

### 10.5.2. Sup to a Mainframe

To use **`sup`** the user must set up a directory named "**`sup`**" on the workstation in the user's home directory. This contains different directories for the various trees that will be backed up, so there will be a "**`Mail`**" directory. This directory will contain two files: "**`crypt`**" and "**`list`**". The "**`crypt`**" file contains one line, terminated with a new line, that contains a single word — an encryption key. "**`list`**" contains one line, terminated with a new line, that contains two words — "**upgrade Mail**".

On the user's mainframe, a file must be created that will be supplied to the **sup** program. It should contain the following line to backup the mail directory:

```
Mail delete host=<workstation> hostbase=/usr/<user> base=/usr/<user> \
crypt=WordInCryptFile login=<user> password=LoginPasswordOnWorkstation
```

Warning: *This file contains the user's password and should be protected appropriately.*

The following Unix shell command issued on the mainframe will backup the mail directory:

```
sup <name of the sup file used in previous paragraph>
```

As a specific example, assume user "**fred**" has a workstation called "**fred**", and his mainframe is the "**gpa**" machine where he has another user account named "**fred**". The password on his workstation is "**purple**". On his workstation, he creates the directory "**/usr/fred/sup/Mail/**" with the two files "**crypt**" and "**list**". The file "**/usr/fred/sup/Mail/crypt**" contains only the encryption key:

```
steppenwolf
```

The file "**/usr/fred/sup/Mail/list**" contains the command to upgrade the "**Mail**" directory:

```
upgrade Mail
```

On the "**gpa**" machine, the file "**/usr/fred/supfile**" contains the following line:

```
Mail delete host=fred hostbase=/usr/fred base=/usr/fred \
crypt=steppenwolf login=fred password=purple
```

This file is protected on "**gpa**", so others cannot see **fred's** password on his workstation.

On the gpa-vax, issuing

```
sup /usr/fred/supfile
```

to the Unix shell will update the MH mail directory from **fred's** workstation deleting any files that exist on the gpa that do not exist on the workstation.

For a more complete description of the features of **sup**, see the *UNIX Workstation Owner's Guide* and *The SUP Software Upgrade Protocol*.

## 10.6. Introduction to Commands and Variables

Unless otherwise specified, any command which prompts for a folder name will offer the user a default. Usually this is MH's idea of the current folder, but sometimes it is the folder name associated with the current buffer if there is one. When prompting for a message, any valid MH message expression may be entered (for example, "**1 3 6**", "**1-3 5 6**", "**unseen**", "**all**"). Unless otherwise specified, a default will be offered (usually the current message).

Some commands mention specific MH utilities, so the user knows how the Hemlock command affects the state of MH and what profile components and special formatting files will be used. Hemlock runs the MH utility programs from a directory indicated by the following variable:

MH Utility Pathname (initial value **"/usr/misc/.mh/bin/"**) [Hemlock *Variable*]

MH utility names are merged with this pathname to find the executable files.

## 10.7. Scanning and Picking Messages

As pointed out in the introduction of this chapter, users typically generate headers or **`scan`** listings of messages with Message Headers, using commands that operate on the messages represented by the headers. Pick Headers (bound to **`h`** in Headers mode) can be used to narrow down (or further select over) the headers in the buffer.

A **`pick`** expression may be entered using either a Lisp syntax or a Unix shell-like/switch notation as described in the MH documentation. The Lisp syntax is as follows:

```
<exp>       ::=  {(not <exp>) | (and <exp>*) | (or <exp>*)
                  | (cc <pattern>) | (date <pattern>)
                  | (from <pattern>) | (search <pattern>)
                  | (subject <pattern>) | (to <pattern>)
                  | (-- <component> <pattern>)
                  | (before <date>) | (after <date>)
                  | (datefield <field>)}

<pattern>   ::=  {<string> | <symbol>}

<component> ::=  {<string> | <symbol>}

<date>      ::=  {<string> | <symbol> | <number>}

<field>     ::=  <string>
```

Anywhere the user enters a **`<symbol>`**, its symbol name is used as a string. Since Hemlock **`read`**s the expression without evaluating it, single quotes ("'") are unnecessary. From the MH documentation,

- A **`<pattern>`** is a Unix **`ed`** regular expression. When using a string to input these, remember that **`\`** is an escape character in Common Lisp.
- A **`<component>`** is a header field name (for example, **reply-to** or **resent-to**).
- A **`<date>`** is an *822*-style specification, a day of the week, "**today**", "**yesterday**", "**tomorrow**", or a number indicating *n* days ago. The *822* standard is basically:

  ```
  dd mmm yy hh:mm:ss zzz
  ```

  which is a two digit day, three letter month (first letter capitalized), two digit year, two digit hour (**`00`** through **`23`**), two digit minute, two digit second (this is optional), and a three letter zone (all capitalized). For example:

  ```
  21 Mar 88 16:00 EST
  ```

- A **`<field>`** is an alternate **`Date:`** field to use with **`(before <date>)`** and **`(after <date>)`** such as **`BB-Posted:`** or **`Delivery-Date:`**.
- Using **`(before <date>)`** and **`(after <date>)`** causes date field parsing, while **`(date <pattern>)`** does string pattern matching.

Since a **`<pattern>`** may be a symbol or string, it should be noted that the symbol name is probably all uppercase characters, and MH will match these only against upper case. MH will match lowercase characters against lower and upper case. Some examples are:

```
;;; All messages to Gripe.
(to "gripe")

;;; All messages to Gripe or about Hemlock.
(or (to "gripe") (subject "hemlock"))

;;; All messages to Gripe with "Hemlock" in the body.
(and (to "gripe") (search "hemlock"))
```

Matching of **<component>** fields is case sensitive, so this example will **pick** over all messages that have been replied to.

```
(or (-- "replied" "") (-- "Replied" ""))
```

MH Scan Line Form (initial value **"library:mh-scan"**) [Hemlock *Variable*]

This is a pathname of a file containing an MH format expression used for header lines.

The header line format must display the message ID as the first non-whitespace item. If the user uses the virtual message deletion feature which is on by default, there must be a space three characters to the right of the message ID. This location is used on header lines to note that a message is flagged for deletion. The second space after the message ID is used for notating answered or replied-to messages.

Message Headers (bound to **C-x r**) [*Command*]

This command prompts for a folder, message (defaulting to "**all**"), and an optional **pick** expression. Typically this will simply be used to generate headers for an entire folder or sequence, and the **pick** expression will not be used. A new Headers buffer is made, and the output of **scan** on the messages indicated is inserted into the buffer. The current window is used, the buffer's point is moved to the first header, and the Headers buffer becomes current. The current value of the Hemlock Fill Column variable is supplied to **scan** as the **-width** switch. The buffer name is set to a string of the form "**Headers <folder> <msgs> <pick expression>**", so the modeline will show what is in the buffer. If no **pick** expression was supplied, none will be shown in the buffer's name. As described in the introduction to this section, the expression may be entered using either a Lisp syntax or a Unix shell-like/switch notation.

MH Lisp Expression (initial value **t**) [Hemlock *Variable*]

When this is set, MH expression prompts are read in a Lisp syntax. Otherwise, the input is of the form of a Unix shell-like/switch notation as described in the MH documentation.

Pick Headers (bound to **h** in Headers mode) [*Command*]

This command is only valid in a Headers buffer. It prompts for a **pick** expression, and the messages shown in the buffer are supplied to **pick** with the expression. The resulting messages are **scan**'ed, deleting the previous contents of the buffer. The current value of Fill Column is used for the **scan**'ing. The buffer's point is moved to the first header. The buffer's name is set to a string of the form "**Headers <folder> <msgs picked over> <pick expression>**", so the modeline will show what is in the buffer. As described in the introduction to this section, the expression may be entered using either a Lisp syntax or a Unix shell-like/switch notation.

Headers Help (bound to **Headers: ?**) [*Command*]

This command displays documentation on Headers mode.

## 10.8. Reading New Mail

Incorporate and Read New Mail (bound to **C-x i** globally and **i** in Headers and Message modes) [*Command*]

This command incorporates new mail into New Mail Folder and creates a Headers buffer with the new messages. An unseen-sequence must be define in the user's MH profile to use this. Any headers generated due to Unseen Headers Message Spec are inserted as well. The buffer's point is positioned on the headers line representing the first unseen message of the newly incorporated mail.

Incorporate New Mail [*Command*]

This command incorporates new mail into New Mail Folder, displaying **inc** output in a pop-up window. This is similar to Incorporate and Read New Mail except that no Headers buffer is generated.

New Mail Folder (initial value **"+inbox"**) [Hemlock *Variable*]

This is the folder into which MH incorporates new mail.

Unseen Headers Message Spec (initial value **nil**) [Hemlock *Variable*]

This is an MH message specification that is suitable for any message prompt. When incorporating new mail and after expunging messages, Hemlock uses this specification in addition to the unseen-sequence name that is taken from the user's MH profile to generate headers for the unseen Headers buffer. This value is a string.

Incorporate New Mail Hook (initial value **nil**) [Hemlock *Variable*]

This is a list of functions which are invoked immediately after new mail is incorporated. The functions should take no arguments.

Store Password (initial value **nil**) [Hemlock *Variable*]

When this is set, the user is only prompted once for his password, and the password is stored for future use.

Authenticate Incorporation (initial value **nil**) [Hemlock *Variable*]
Authentication User Name (initial value **nil**) [Hemlock *Variable*]

When Authenticate Incorporation is set, incorporating new mail prompts for a password to access a remote mail-drop.

When incorporating new mail accesses a remote mail-drop, Authentication User Name is the user name supplied for authentication on the remote machine. If this is **nil**, Hemlock uses the local name.

## 10.9. Reading Messages

This section describes basic commands that show the current, next, and previous messages, as well as a couple advanced commands. Show Message (bound to **SPACE** in Headers mode) will display the message represented by the **scan** line the Hemlock cursor is on. Deleted messages are considered special, and the more conveniently bound commands for viewing the next and previous messages (Next Undeleted Message bound to **n** and Previous Undeleted Message bound to **p**, both in Headers and Message modes) will ignore them. Next Message and Previous Message (bound to **M-n** and **M-p** in Headers and Message modes) may be invoked if reading a message is desired regardless of whether it has been deleted.

Show Message (bound to **SPACE** and **.** in Headers mode) [*Command*]

This command, when invoked in a Headers buffer, displays the current message (the message the cursor is on), by replacing any previous message that has not been preserved with Keep Message. The current message is also removed from the unseen sequence. The Message buffer becomes the current buffer using the current window. The buffer's point will be moved to the beginning of the buffer, and the buffer's name will be set to a string of the form "**Message <folder> <msg-id>**".

The Message buffer is read-only and may not be modified. The command Goto Headers Buffer issued in the Message buffer makes the associated Headers buffer current.

When not in a Headers buffer, this command prompts for a folder and message. A unique Message buffer is obtained, and its name is set to a string of the form "**`Message <folder> <msg-id>`**". The buffer's point is moved to the beginning of the buffer, and the current window is used to display the message.

Specifying multiple messages inserts all the messages into the same buffer. If the user wishes to show more than one message, it is expected that he will generate a headers buffer with the intended messages, and then use the message sequencing commands described below.

Next Message (bound to **M-n** in Headers and Message modes) [*Command*]

This command is only meaningful in a Headers buffer or a Message buffer associated with a Headers buffer. In a Headers buffer, the point is moved to the next message, and if there is one, it is shown as described in the Show Message command.

In a Message buffer, the message after the currently visible message is displayed. This clobbers the buffer's contents. Note, if the Message buffer is associated with a Draft buffer, invoking this command breaks that association. Using Keep Message preserves the Message buffer and any association with a Draft buffer.

The Message buffer's name is set as described in the Show Message command.

Previous Message (bound to **M-p** in Headers and Message modes) [*Command*]

This command is only meaningful in a Headers buffer or a Message buffer associated with a Headers buffer. In a Headers buffer, the point is moved to the previous message, and if there is one, it is shown as described in the Show Message command.

In a Message buffer, the message before the currently visible message is displayed. This clobbers the buffer's contents. Note, if the Message buffer is associated with a Draft buffer, invoking this command breaks that association. Using Keep Message preserves the Message buffer and any association with a Draft buffer.

The Message buffer's name is set as described in the Show Message command.

Next Undeleted Message (bound to **n** in Headers and Message modes) [*Command*]

This command is only meaningful in a Headers buffer or a Message buffer associated with a Headers buffer. In a Headers buffer, the point is moved to the next undeleted message, and if there is one, it is shown as described in the Show Message command.

In a Message buffer, the first undeleted message after the currently visible message is displayed. This clobbers the buffer's contents. Note, if the Message buffer is associated with a Draft buffer, invoking this command breaks that association. The Keep Message command preserves the Message buffer and any association with a Draft buffer.

The Message buffer's name is set as described in the Show Message command.

Previous Undeleted Message (bound to **p** in Headers and Message modes) [*Command*]

This command is only meaningful in a Headers buffer or a Message buffer associated with a Headers buffer. In a Headers buffer, the point is moved to the previous undeleted message, and if there is one, it is shown as described in the Show Message command.

In a Message buffer, the first undeleted message before the currently visible message is displayed. This clobbers the buffer's contents. Note, if the Message buffer is associated with a Draft buffer, invoking this command breaks that association. The Keep Message command preserves the Message buffer and any association with a Draft buffer.

The Message buffer's name is set as described in the Show Message command.

Scroll Message (bound to **SPACE** and **C-v** in Message mode) [*Command*]
Scroll Message Showing Next (initial value `t`) [Hemlock *Variable*]

This command scrolls the current window down through the current message. If the end of the message is visible and Scroll Message Showing Next is not `nil`, then show the next undeleted message.

Keep Message [*Command*]

This command can only be invoked in a Message buffer. It causes the Message buffer to continue to exist when the user invokes commands to view other messages either within the kept Message buffer or its associated Headers buffer. This is useful for getting two messages into different buffers. It is also useful for retaining Message buffers which would otherwise be deleted when an associated draft message is delivered.

Message Help (bound to **Message: ?**) [*Command*]

This command displays documentation on Message mode.

## 10.10. Sending Messages

The most useful commands for sending mail are Send Message (bound to **m** and **s** in Headers and Message modes), Reply to Message (bound to **r** in Headers mode), and Reply to Message in Other Window (bound to **r** in Message mode). These commands set up a Draft buffer and associate a Message buffer with the draft when possible. To actually deliver the message to its recipient(s), use Deliver Message (bound to **H-s** in Draft mode). To abort sending mail, use Delete Draft and Buffer (bound to **H-q** in Draft mode). If one wants to temporarily stop composing a draft with the intention of finishing it later, then the Save File command (bound to **C-x C-s**) will save the draft to the user's draft folder.

Draft buffers have a special Hemlock minor mode called Draft mode. The major mode of a Draft buffer is taken from the Default Modes variable. The user may wish to arrange that Text mode (and possibly Fill mode or Save mode) be turned on whenever Draft mode is set. For a further description of how to manipulate modes in Hemlock see the *Hemlock Command Implementor's Manual*.

Send Message (bound to **s** and **m** in Headers and Message modes and **C-x m** globally) [*Command*]

This command, when invoked in a Headers buffer, creates a unique Draft buffer and a unique Message buffer. The current message is inserted in the Message buffer, and the Draft buffer is displayed in the current window. The Draft buffer's point is moved to the end of the line containing `To:` if it exists. The name of the draft message file is used to produce the buffer's name. A pathname is associated with the Draft buffer so that Save File can be used to incrementally save a composition before delivering it. The `comp` utility will be used to allocate a draft message in the user's MH draft folder and to insert the proper header components into the draft message. Both the Draft and Message buffers are associated with the Headers buffer, and the Draft buffer is associated with the Message buffer.

When invoked in a Message buffer, a unique Draft buffer is created, and these two buffers are associated. If the Message buffer is associated with a Headers buffer, this association is propagated to the Draft buffer. Showing other messages while in this Headers buffer will not affect this Message buffer.

When not in a Headers or Message buffer, this command does the same thing as described in the previous two cases, but there are no Message or Headers buffer manipulations.

Deliver Message will deliver the draft to its intended recipient(s).

The Goto Headers Buffer command, when invoked in a Draft or Message buffer, makes the associated Headers buffer current. The Goto Message Buffer command, when invoked in a Draft buffer, makes the associated Message buffer current.

Reply to Message (bound to **r** in Headers mode) [*Command*]
Reply to Message in Other Window (bound to **r** in Message mode) [*Command*]
Reply to Message Prefix Action [Hemlock *Variable*]

Reply to Message, when invoked in a Headers buffer, creates a unique Draft buffer and a unique Message buffer. The current message is inserted in the Message buffer, and the Draft buffer is displayed in the current window. The draft components are set up in reply to the message, and the Draft buffer's point is moved to the end of the buffer. The name of the draft message file is used to produce the buffer's name. A pathname is associated with the Draft buffer so that Save File can be used to incrementally save a composition before delivering it. The **`repl`** utility will be used to allocate a draft message file in the user's MH draft folder and to insert the proper header components into the draft message. Both the Draft and Message buffers are associated with the Headers buffer, and the Draft buffer is associated with the Message buffer.

When invoked in a Message buffer, a unique Draft buffer is set up using the message in the buffer as the associated message. Any previous association between the Message buffer and a Draft buffer is removed. Any association of the Message buffer with a Headers buffer is propagated to the Draft buffer.

When not in a Headers buffer or Message buffer, this command prompts for a folder and message to reply to. This message is inserted into a unique Message buffer, and a unique Draft buffer is created as in the previous two cases. There is no association of either the Message buffer or the Draft buffer with a Headers buffer.

When a prefix argument is supplied, Reply to Message Prefix Action is considered with respect to supplying carbon copy switches to **`repl`**. This variable's value is one of **:cc-all**, :**no-cc-all**, or **`nil`**. See section 10.18 for examples of how to use this.

Reply to Message in Other Window is identical to Reply to Message, but the current window is split showing the Draft buffer in the new window. The split window displays the Message buffer.

Deliver Message will deliver the draft to its intended recipient(s).

The Goto Headers Buffer commmand, when invoked in a Draft or Message buffer, makes the associated Headers buffer current. The Goto Message Buffer command, when invoked in a Draft buffer, makes the associated Message buffer current.

Forward Message (bound to **f** in Headers and Message modes) [*Command*]

This command, when invoked in a Headers buffer, creates a unique Draft buffer. The current message is inserted in the draft by using the **`forw`** utility, and the Draft buffer is shown in the current window. The name of the draft message file is used to produce the buffer's name. A pathname is associated with the Draft buffer so that Save File can be used to incrementally save a composition before delivering it. The Draft buffer is associated with the Headers buffer, but no Message buffer is created since the message is already a part of the draft.

When invoked in a Message buffer, a unique Draft buffer is set up inserting the message into the Draft buffer. The Message buffer is not associated with the Draft buffer because the message is already a part of the draft. However, any association of the Message buffer with a Headers buffer is propagated to the Draft buffer.

When not in a Headers buffer or Message buffer, this command prompts for a folder and message to forward. A Draft buffer is created as described in the previous two cases.

Deliver Message will deliver the draft to its intended recipient(s).

Deliver Message (bound to **H-s** and **H-c** in Draft mode) [*Command*]
Deliver Message Confirm (initial value **`nil`**) [Hemlock *Variable*]

This command, when invoked in a Draft buffer, saves the file and uses the MH **`send`** utility to deliver the draft. If the draft is a reply to some message, then **`anno`** is used to annotate that message with a "**`replied`**" component. Any Headers buffers containing the replied-to message are updated with an "**A**" placed in the appropriate headers line two characters after the message ID. Before doing any of this, confirmation is asked for based on Deliver Message Confirm.

When not in a Draft buffer, this prompts for a draft message ID and invokes **`send`** on that draft message to deliver it. Sending a draft in this way severs any association that draft may have had with a message being replied to, so no annotation will occur.

Delete Draft and Buffer (bound to **H-q** in Draft mode) [*Command*]

This command, when invoked in a Draft buffer, deletes the draft message file and the buffer. This also deletes any associated message buffer unless the user preserved it with Keep Message.

Remail Message (bound to **H-r** in Headers and Message modes) [*Command*]

This command, when invoked in a Headers or Message buffer, prompts for resend **`To:`** and resend **`Cc:`** addresses, remailing the current message. When invoked in any other kind of buffer, this command prompts for a folder and message as well. MH's **`dist`** sets up a draft folder message which is then modified. The above mentioned addresses are inserted on the **`Resent-To:`** and **`Resent-Cc:`** lines. Then the message is delivered.

There is no mechanism for annotating messages as having been remailed.

Draft Help (bound to **Draft: H-?**) [*Command*]

This command displays documentation on Draft mode.

## 10.11. Convenience Commands for Message and Draft Buffers

This section describes how to switch from a Message or Draft buffer to its associated Headers buffer, or from a Draft buffer to its associated Message buffer. There are also commands for various styles of inserting text from a Message buffer into a Draft buffer.

Goto Headers Buffer (bound to **^** in Message mode and **H-^** in Draft mode) [*Command*]

This command, when invoked in a Message or Draft buffer with an associated Headers buffer, places the associated Headers buffer in the current window.

The cursor is moved to the headers line of the associated message.

Goto Message Buffer (bound to **H-m** in Draft mode) [*Command*]

This command, when invoked in a Draft buffer with an associated Message buffer, places the associated Message buffer in the current window.

Insert Message Region (bound to **H-y** in appropriate modes) [*Command*]
Message Insertion Prefix (initial value **"   "**) [Hemlock *Variable*]
Message Insertion Column (initial value **75**) [Hemlock *Variable*]

This command, when invoked in a Message or News-Message (where it is bound) buffer that has an associated Draft or Post buffer, copies the current active region into the Draft or Post buffer. It is filled using Message Insertion Prefix (which defaults to three spaces) and Message Insertion Column. If an argument is supplied, the filling is inhibited.

Insert Message Buffer (bound to **H-y** in appropriate modes) [*Command*]
Message Buffer Insertion Prefix (initial value **"    "**) [Hemlock *Variable*]

This command, when invoked in a Draft or Post (where it is bound) buffer with an associated Message or News-Message buffer, or when in a Message (or News-Message) buffer that has an associated Draft buffer, inserts the Message buffer into the Draft (or Post) buffer. Each inserted line is modified by prefixing it with Message Buffer Insertion Prefix (which defaults to four spaces) . If an argument is supplied, the prefixing is inhibited.

Edit Message Buffer (bound to **e** in Message mode) [*Command*]

This command puts the current Message buffer in Text mode and makes it writable (Message buffers are normally read-only). The pathname of the file which the message is in is associated with the buffer making saving possible. A recursive edit is entered, and the user is allowed to make changes to the message. When the recursive edit is exited, if the buffer is modified, the user is asked if the changes should be saved. The buffer is marked unmodified, and the pathname is disassociated from the buffer. The buffer otherwise returns to its previous state as a Message buffer. If the recursive edit is aborted, the user is not asked to save the file, and the buffer remains changed though it is marked unmodified.

## 10.12. Deleting Messages

The main command described in this section is Headers Delete Message (bound to **k** in Headers and Message modes). A useful command for reading new mail is Delete Message and Show Next (bound to **d** in Message mode) which deletes the current message and shows the next undeleted message.

Since messages are by default deleted using a virtual message deletion mechanism, Expunge Messages (bound to **!** in Headers mode) should be mentioned here. This is described in section 10.16.

Virtual Message Deletion (initial value **t**) [Hemlock *Variable*]

When set, Delete Message adds a message to the "**hemlockdeleted**" sequence; otherwise, **rmm** is invoked on the message immediately.

Delete Message [*Command*]

This command prompts for a folder, messages, and an optional **pick** expression. When invoked in a Headers buffer of the specified folder, the prompt for a message specification will default to the those messages in that Headers buffer.

When the variable Virtual Message Deletion is set, this command merely flags the messages for deletion by adding them to the "**hemlockdeleted**" sequence. Then this updates any Headers buffers representing the folder. It notates each headers line referring to a deleted message with a "**D**" in the third character position after the message ID.

When Virtual Message Deletion is not set, **rmm** is invoked on the message, and each headers line referring to the deleted message is deleted from its buffer

Headers Delete Message (bound to **k** in Headers and Message modes) [*Command*]

This command, when invoked in a Headers buffer, deletes the message on the current line as described in Delete Message.

When invoked in a Message buffer, the message displayed in it is deleted as described in Delete Message.

Delete Message and Show Next (bound to **k** in Headers and Message modes) [*Command*]

This command is only valid in a Headers buffer or a Message buffer associated with some Headers buffer. The current message is deleted as with the Delete Message command. Then the next message is shown as with Next Undeleted Message.

Delete Message and Down Line (bound to **d** in Headers mode) [*Command*]

This command, when invoked in a Headers buffer, deletes the message on the current line. Then the point is moved to the next non-blank line.

Undelete Message [*Command*]

This command is only meaningful when Virtual Message Deletion is set. This prompts for a folder, messages, and an optional **pick** expression. When in a Headers buffer of the specified folder, the messages prompt defaults to those messages in the buffer. All Headers buffers representing the folder are updated. Each headers line referring to an undeleted message is notated by replacing the "**D**" in the third character position after the message ID with a space.

Headers Undelete Message (bound to **u** in Headers and Message modes) [*Command*]

This command is only meaningful when Virtual Message Deletion is set. When invoked in a Headers buffer, the message on the current line is undeleted as described in Undelete Message.

When invoked in a Message buffer, the message displayed in it is undeleted as described in Undelete Message.

## 10.13. Folder Operations

List Folders [*Command*]

This command displays a list of all current mail folders in the user's top-level mail directory in a Hemlock pop-up window.

Create Folder [*Command*]

This command prompts for and creates a folder. If the folder already exists, an error is signaled.

Delete Folder [*Command*]

This command prompts for a folder and uses **rmf** to delete it. Note that no confirmation is asked for.

## 10.14. Refiling Messages

Refile Message [*Command*]

This command prompts for a folder, messages, an optional **pick** expression, and a destination folder. When invoked in a Headers buffer of the specified folder, the message prompt offers a default of those messages in the buffer. If the destination folder does not exist, the user is asked to create it. The resulting messages are refiled with the **refile** utility. All Headers buffers for the folder are updated. Each line referring to a refiled message is deleted from its buffer.

Headers Refile Message (bound to **o** in Headers and Message modes) [*Command*]

This command, when invoked in a Headers buffer, prompts for a destination folder, refiling the message on the current line with **refile**. If the destination folder does not exist, the user is asked to create it. Any Headers buffers containing messages for that folder are updated. Each headers line referring to the refiled message is deleted from its buffer.

When invoked in a Message buffer, that message is refiled as described above.

## 10.15. Marking Messages

Mark Message [*Command*]

This command prompts for a folder, message, and sequence and adds (deletes) the message specification to (from) the sequence. By default this adds the message, but if an argument is supplied, this deletes the message. When invoked in a Headers buffer or Message buffer, this only prompts for a sequence and uses the current message.

## 10.16. Terminating Headers Buffers

The user never actually *exits* the mailer. He can leave mail buffers lying around while conducting other editing tasks, selecting them and continuing his mail handling whenever. There still is a need for various methods of terminating or cleaning up Headers buffers. The two most useful commands in this section are Expunge Messages and Quit Headers.

Expunge Messages Confirm (initial value **t**) [Hemlock *Variable*]

When this is set, Quit Headers and Expunge Messages will ask for confirmation before expunging messages and packing the folder's message ID's.

Temporary Draft Folder (initial value **nil**) [Hemlock *Variable*]

This is a folder name where MH **fcc:** messages are kept with the intention that this folder's messages will be deleted and expunged whenever messages from any folder are expunged (for example, when Expunge Messages or Quit Headers is invoked.

Expunge Messages (bound to **!** in Headers mode) [*Command*]

This command deletes messages **mark**'ed for deletion, and compacts the folder's message ID's. If there are messages to expunge, ask the user for confirmation, displaying the folder name. This can be inhibited by setting Expunge Messages Confirm to **nil**. When Temporary Draft Folder is not **nil**, this command deletes and expunges that folder's messages regardless of the folder in which the user invokes it, and a negative response to the request for confirmation inhibits this.

When invoked in a Headers buffer, the messages in that folder's "**hemlockdeleted**" sequence are deleted by invoking **rmm**. Then the ID's of the folder's remaining messages are compacted using the **folder** utility. Since headers must be regenerated due to renumbering or reassigning message ID's, and because Headers buffers become inconsistent after messages are deleted, Hemlock must regenerate all the headers for the folder. Multiple Headers buffers for the same folder are then collapsed into one buffer, deleting unnecessary duplicates. Any Message buffers associated with these Headers buffers are deleted.

If there is an unseen Headers buffer for the folder, it is handled separately from the Headers buffers described above. Hemlock tries to update it by filling it only with remaining unseen message headers. Additionally, any headers generated due to Unseen Headers Message Spec are inserted. If there are no headers, unseen or otherwise, the buffer is left blank.

Any Draft buffer set up as a reply to a message in the folder is affected as well since the associated message has possibly been deleted. When a draft of this type is delivered, no message will be annotated as having been replied to.

When invoked in a Message buffer, this uses its corresponding folder as the folder argument. The same updating as described above occurs.

In any other type of buffer, a folder is prompted for.

Quit Headers (bound to **q** in Headers and Message modes) [*Command*]

This command affects the current Headers buffer. When there are deleted messages, ask the user for confirmation on expunging the messages and packing the folder's message ID's. This prompting can be inhibited by setting Expunge Messages Confirm to **nil**. After deleting and packing, this deletes the buffer and all its associated Message buffers.

Other Headers buffers regarding the same folder are handled as described in Expunge Messages, but the buffer this command is invoked in is always deleted.

When Temporary Draft Folder is not **nil**, this folder's messages are deleted and expunged regardless of the folder in which the user invokes this command. A negative response to the above mentioned request for confirmation inhibits this.

Delete Headers Buffer and Message Buffers [*Command*]

This command prompts for a Headers buffer to delete along with its associated Message buffers. Any associated Draft buffers are left intact, but their corresponding Message buffers will be deleted. When invoked in a Headers buffer or a Message buffer associated with a Headers buffer, that Headers buffer is offered as a default.

## 10.17. Miscellaneous Commands

List Mail Buffers (bound to **l** in Headers and Message modes **H-l** in Draft mode) [*Command*]

This command shows a list of all mail Message, Headers, and Draft buffers.

If a Message buffer has an associated Headers buffer, it is displayed to the right of the Message buffer's name.

If a Draft buffer has an associated Message buffer, it is displayed to the right of the Draft buffer's name. If a Draft buffer has no associated Message buffer, but it is associated with a Headers buffer, then the name of the Headers buffer is displayed to the right of the Draft buffer.

For each buffer listed, if it is modified, then an asterisk is displayed before the name of the buffer.

## 10.18. Styles of Usage

This section discusses some styles of usage or ways to make use of some of the features of Hemlock's interface to MH that might not be obvious. In each case, setting some variables and/or remembering an extra side effect of a command will lend greater flexibility and functionality to the user.

### 10.18.1. Unseen Headers Message Spec

The unseen Headers buffer by default only shows unseen headers which is adequate for one folder, simple mail handling. Some people use their New Mail Folder only for incoming mail, refiling or otherwise dispatching a message immediately. Under this mode it is easy to conceive of the user not having time to respond to a message, but he would like to leave it in this folder to remind him to take care of it. Using the Unseen Headers Message Spec variable, the user can cause all the messages the New Mail Folder to be inserted into the unseen Headers buffer whenever just unseen headers would be. This way he sees all the messages that require immediate attention.

To achieve the above effect, Unseen Headers Message Spec should be set to the string **`"all"`**. This variable can be set to any general MH message specification (see section 10.6 of this chapter), so the user can include headers of messages other than those that have not been seen without having to insert all of them. For example, the user could set the variable to **`"flagged"`** and use the Mark Message command to add messages he's concerned about to the **`"flagged"`** sequence. Then the user would see new mail and interesting mail in his unseen Headers buffer, but he doesn't have to see everything in his New Mail Folder.

### 10.18.2. Temporary Draft Folder

Section 10.4.2 of this chapter discusses how to make MH keep personal copies of outgoing mail. The method described will cause a copy of every outgoing message to be saved forever and requires the user to go through his **`Fcc:`** folder, weeding out those he does not need. The Temporary Draft Folder variable can name a folder whose messages will be deleted and expunged whenever any folder's messages are expunged. By naming this folder in the MH profile and components files, copies of outgoing messages can be saved temporarily. They will be cleaned up automatically, but the user still has a time frame in which he can permanently save a copy of an outgoing message. This folder can be visited with Message Headers, and messages can be refiled just like any other folder.

### 10.18.3. Reply to Message Prefix Action

Depending on the kinds of messages one tends to handle, the user may find himself usually replying to everyone who receives a certain message, or he may find that this is only desired occasionally. In either case, the user can set up his MH profile to do one thing by default, using the Reply to Message Prefix Action variable in combination with a prefix argument to the Reply to Message command to get the other effect.

For example, the following line in one's MH profile will cause MH to reply to everyone receiving a certain message (except for the user himself since he saves personal copies with the **`-fcc`** switch):

```
repl: -cc all -nocc me -fcc out-copy
```

This user can set Reply to Message Prefix Action to be **`:no-cc-all`**. Then whenever he invokes Reply to Message with a prefix argument, instead of replying to everyone, the draft will be set up in reply only to the person who sent the mail.

As an alternative example, not specifying anything in one's MH profile and setting this variable to **`:cc-all`** will have a default effect of replying only to the sender of a piece of mail. Then invoking Reply to Message with a prefix argument will cause everyone who received the mail to get a copy of the reply. If the user does not want a **`cc:`** copy, then he can add **`-nocc me`** as a default switch and value in his MH profile.

## 10.19. Wallchart

**Global bindings:**

| | |
|---|---|
| Incorporate and Read New Mail | **C-x i** |
| Send Message | **C-x m** |
| Message Headers | **C-x r** |

**Headers and Message modes bindings:**

| | |
|---|---|
| Next Undeleted Message | **n** |
| Previous Undeleted Message | **p** |
| Send Message | **s**, **m** |
| Forward Message | **f** |
| Headers Delete Message | **k** |
| Headers Undelete Message | **u** |
| Headers Refile Message | **o** |
| List Mail Buffers | **l** |
| Quit Headers | **q** |
| Incorporate and Read New Mail | **i** |
| Next Message | **M-n** |
| Previous Message | **M-p** |
| Beginning of Buffer | **<** |
| End of Buffer | **>** |

**Headers mode bindings:**

| | |
|---|---|
| Delete Message and Down Line | **d** |
| Pick Headers | **h** |
| Show Message | **space**, **.** |
| Reply to Message | **r** |
| Expunge Messages | **!** |

**Message mode bindings:**

| | |
|---|---|
| Delete Message and Show Next | **d** |
| Goto Headers Buffer | **^** |
| Scroll Message | **space** |

| | |
|---|---|
| Scroll Message | **C-v** |
| Scroll Window Up | **backspace**, **delete** |
| Reply to Message in Other Window | **r** |
| Edit Message Buffer | **e** |
| Insert Message Region | **H-y** |

**Draft mode bindings:**

| | |
|---|---|
| Goto Headers Buffer | **H-^** |
| Goto Message Buffer | **H-m** |
| Deliver Message | **H-s**, **H-c** |
| Insert Message Buffer | **H-y** |
| Delete Draft and Buffer | **H-q** |
| List Mail Buffers | **H-l** |

# Chapter 11

# The Hemlock Netnews Interface

## 11.1. Introduction to Netnews in Hemlock

Hemlock provides a facility for reading bulletin boards through the NetNews Transfer Protocol (NNTP). You can easily read Netnews, reply to news posts, post messages, etc. The news reading interface is consistent with that of the Hemlock mailer, and most Netnews commands function in the same manner as their mailer counterparts.

Netnews can be read in one of two different modes. The first mode, invoked by the Netnews command, allows the user to read new messages in groups which the user has specified. This method of reading netnews will track the highest numbered message in each newsgroup and only show new messages which have arrived since then. The Netnews Browse command invokes the other method of reading netnews. This mode displays a list of all newsgroups, and the user may choose to read messages in any of them. By default, the news reader will not track the latest message read when browsing, and it will always display the last few messages.

## 11.2. Setting Up Netnews

To start reading bulletin boards from Hemlock you probably need to create a file containing the newsgroups you want to read.

Netnews Group File (initial value **`".hemlock-groups"`**) [Hemlock *Variable*]

When you invoke the Netnews command, Hemlock merges the value of this variable with your home directory and looks there for a list of groups (one per line) to read.

Netnews Database File (initial value **`".hemlock-netnews"`**) [Hemlock *Variable*]

When you invoke the Netnews command, Hemlock merges the value of this variable with your home directory. This file maintains a pointer to the highest numbered message read in each group in Netnews Group File.

List All Groups [*Command*]

When you invoke this command, Hemlock creates a buffer called Netnews Groups and inserts the names of all accessible Netnews groups into it alphabetically. You may find this useful if you choose to set up your Netnews Group File manually.

Netnews NNTP Server (initial value **`"netnews.srv.cs.cmu.edu"`**) [Hemlock *Variable*]

This variable stores the host name of the machine which Hemlock will use as the NNTP server.

Netnews NNTP Timeout Period (initial value **30**) [Hemlock *Variable*]

This is the number of seconds Hemlock will wait trying to connect to the NNTP server. If a connection is not made within this time period, the connection will time out and an error will be signalled.

### 11.2.1. News-Browse Mode

News-Browse mode provides an easy method of adding groups to your Netnews Group File.

Netnews Browse [*Command*]

This command sets up a buffer in News-Browse mode with all available groups listed one per line. Groups may be read or added to your group file using various commands in this mode.

Netnews Browse Add Group To File (bound to **a** in News-Browse mode) [*Command*]
Netnews Browse Pointer Add Group to File [*Command*]

Netnews Browse Add Group to File adds the group under the point to your group file, and Netnews Browse Pointer Add Group To File adds the group under the mouse pointer without moving the point.

Netnews Browse Read Group (bound to **space** in News-Browse mode) [*Command*]
Netnews Browse Pointer Read Group [*Command*]

Netnews Browse Read Group and Netnews Browse Pointer Read Group read the group under the cursor and the group under the mouse pointer, respectively. These commands neither use nor modify the contents of your Netnews Database File; they will always present the last few messages in the newsgroup, regardless of the last message read. Netnews Browse Pointer Read Group does not modify the position of the point.

Netnews Quit Browse [*Command*]

This command exits News-Browse mode.

The Next Line and Previous Line commands are conveniently bound to **n** and **p** in this mode.

## 11.3. Starting Netnews

Once your Netnews Group File is set up, you may begin reading netnews.

Netnews [*Command*]

This command is the main entry point for reading bulletin boards in Hemlock. Without an argument, the system looks for what bulletin boards to read in the value of Netnews Group File and reads each of them in succession. Hemlock keeps a pointer to the last message you read in each of these groups in your Netnews Database File. Bulletin boards may be added to your Netnews Group File manually or by using the Netnews Browse facility. With an argument, Hemlock prompts the user for the name of a bulletin board and reads it.

Netnews Look at Group [*Command*]

This command prompts for a group and reads it, ignoring the information in your Netnews Database File.

When you read a group, Hemlock creates a buffer that contains important header information for the messages in that group. There are four fields in each header, one each for the *date*, *lines*, *from*, and *subject*. The *date* field shows when the message was sent, the *lines* field displays how long the message is in lines, the *from* field shows who sent the message, and the *subject* field displays the subject of this message. If a field for a message is not available, **NA** will appear instead. You may alter the length of each of these fields by modifying the following Hemlock variables:

Netnews Before Date Field Pad (initial value **1**) [Hemlock *Variable*]

How many spaces should be inserted before the date in News-Headers buffers.

Netnews Date Field Length (initial value **6**) [Hemlock *Variable*]
Netnews Line Field Length (initial value **3**) [Hemlock *Variable*]
Netnews From Field Length (initial value **20**) [Hemlock *Variable*]
Netnews Subject Field Length (initial value **43**) [Hemlock *Variable*]

These variables control how long the *date*, *line*, *from*, and *subject* fields should be in News-Headers buffers.

Netnews Field Padding (initial value **2**) [Hemlock *Variable*]

How many spaces should be left between the Netnews *date*, *from*, *lines*, and *subject* fields after padding to the required length.

For increased speed, Hemlock only inserts headers for a subset of the messages in each group. If you have never read a certain group, and the value of Netnews New Group Style is **:from-end** (the default), Hemlock inserts some number of the last messages in the group, determined by the value of Netnews Batch Count. If the value of Netnews New Group Style is **:from-start**, Hemlock will insert the first batch of messages in the group. If you have read a group before, Hemlock will insert the batch of messages following the highest numbered message that you had read previously.

Netnews Start Over Threshold (initial value **350**) [Hemlock *Variable*]

If the number of new messages in a group exceeds the value of this variable and Netnews New Group Style is **:from-end**, Hemlock asks if you would like to start reading this group from the end.

You may at any time go beyond the messages that are visible using the Netnews Next Line, Netnews Previous Line, Netnews Headers Scroll Window Up, and Netnews Headers Scroll Down commands in News-Headers mode, or the Netnews Next Article and Netnews Previous Article commands in News-Message mode.

Netnews Fetch All Headers (initial value **nil**) [Hemlock *Variable*]

This variable determines whether Netnews will fetch all headers immediately upon entering a new group.

Netnews Batch Count (initial value **50**) [Hemlock *Variable*]

This variable determines how many headers the Netnews facility will fetch at a time.

Netnews New Group Style (initial value **:from-end**) [Hemlock *Variable*]

This variable determines what happens when you read a group that you have never read before. When it is **:from-start**, the Netnews command will read from the beginning of a new group forward. When it is **:from-end**, the default, Netnews will read the group from the end backward.

## 11.4. Reading Messages

From a News-Headers buffer, you may read messages, reply to messages via the Hemlock mailer, or reply to messages via post. Some commands are also bound to ease getting from one header to another.

Netnews Show Article (bound to **space** in News-Headers mode) [*Command*]
Netnews Read Style (initial value **`:multiple`**) [Hemlock *Variable*]
Netnews Headers Proportion (initial value **`0.25`**) [Hemlock *Variable*]

This command puts the body of the message header under the current point into a News-Message buffer. If the value of Netnews Read Style is **`:single`**, Hemlock changes to the News-Message buffer. If it is **`:multiple`**, then Hemlock splits the current window into two windows, one for headers and one for message bodies. The headers window takes up a proportion of the current window based on the value of Netnews Headers Proportion. If the window displaying the News-Headers buffer has already been split, and the message currently displayed in the News-Message window is the same as the one under the current point, this command behaves just like Netnews Message Scroll Down.

Netnews Message Header Fields (initial value **`nil`**) [Hemlock *Variable*]

When this variable is **`nil`**, all available fields are displayed in the header of a message. Otherwise, this variable should containt a list of fields to include in message headers. If an element of this list is an atom, then it should be the string name of a field. If it is a cons, then the car should be the string name of a field, and the cdr should be the length to which this field should be limited. Any string name is acceptable, and fields that do not exist are ignored.

Netnews Show Whole Header (bound to **w** in News-Headers and News-Message modes.) [*Command*]

This command displays the entire header for the message currently being read. This is to undo the effects of Netnews Message Header Fields for the current message.

Netnews Next Line (bound to **C-n** and **Downarrow** in News-Headers mode) [*Command*]
Netnews Last Header Style (initial value **`:next-headers`**) [Hemlock *Variable*]

This command moves the current point to the next line. If you are on the last visible message, and there are more in the current group, headers for these messages will be inserted. If you are on the last header and there are no more messages in this group, then Hemlock will take some action based on the value of Netnews Last Header Style. If the value of this variable is **`:feep`**, Hemlock feeps you indicating there are no more messages. If the value is **`:next-headers`**, Hemlock reads in the headers for the next group in your Netnews Group File. If the value is **`:next-article`**, Hemlock goes on to the next group and shows you the first unread message.

Netnews Previous Line (bound to **C-p** and **Uparrow** in News-Headers mode) [*Command*]

This command moves the current point to the previous line. If you are on the first visible header, and there are more previous messages, Hemlock inserts the headers for these messages.

Netnews Headers Scroll Window Down (bound to **C-v** in News-Headers mode) [*Command*]
Netnews Headers Scroll Window Up (bound to **M-v** in News-Headers mode) [*Command*]

These commands scroll the headers window up or down one screenfull. If the end of the buffer is visible, Hemlock inserts the next batch of headers.

Netnews Next Article (bound to **n** in News-Message and News-Headers modes) [*Command*]
Netnews Previous Article (bound to **p** in News-Message and News-Headers modes) [*Command*]

These commands insert the next or previous message into a message buffer.

Netnews Message Scroll Down (bound to **space** in News-Message mode) [*Command*]
Netnews Scroll Show Next Message (initial value **`t`**) [Hemlock *Variable*]

If the end of the current message is visible, Hemlock feeps the user if the value of Netnews Scroll Show Next Message is non-**`nil`**, or it inserts the next message into this message buffer if that variable is **`nil`**. If the end of the message is not visible, then Hemlock shows the next screenfull of the current message.

Netnews Message Quit (bound to **q** in News-Message mode) [*Command*]

This command deletes the current message buffer and makes the associated News-Headers buffer current.

Netnews Goto Headers Buffer (bound to **H-h** in News-Message mode) [*Command*]

This command, when invoked from a News-Message buffer with an associated News-Headers buffer, places the associated News-Headers buffer into the current window.

Netnews Message Keep Buffer (bound to **k** in News-Message mode) [*Command*]

By default, Hemlock uses one buffer to display all messages in a group, one at a time. This command tells Hemlock to keep the current message buffer intact and start reading messages in another buffer.

Netnews Select Message Buffer (bound to **H-m** in News-Headers and Post modes.) [*Command*]

In News-Headers mode, this command selects the buffer containing the last message read. In Post mode, it selects the associated News-Message buffer, if there is one.

Netnews Append to File (bound to **a** in News-Headers and News-Message modes.) [*Command*]
Netnews Message File (initial value **`"netnews-messages.txt"`**) [Hemlock *Variable*]

This command prompts for a file which the current message will be appended to. The default file is the value of Netnews Message File merged with your home directory.

Netnews Headers File Message (bound to **o** in News-Headers mode) [*Command*]

This command prompts for a mail folder and files the message under the point into it. If the folder does not exist, Hemlock will ask if it should be created.

Netnews Message File Message (bound to **o** in News-Message mode) [*Command*]

This command prompts for a mail folder and files the current message there. If the folder does not exist, Hemlock will ask if it should be created.

Fetch All Headers (bound to **f** in Netnews Headers mode) [*Command*]

In a forward reading Netnews headers buffer, this command inserts all headers after the last visible one into the headers buffer. If Hemlock is reading this group backward, the system inserts all headers before the first visible one into the headers buffer.

Netnews Go to Next Group (bound to **g** in News-Headers and News-Message modes.) [*Command*]

This command goes to the next group in your Netnews Group File. Before going on, it sets the group pointer in Netnews Database Filename to the last message you read. With an argument, the command does not modify the group pointer for the current group.

Netnews Quit Starting Here (bound to **.** in News-Headers and News-Message modes) [*Command*]

This command goes to the next group in your Netnews Group File, setting the netnews pointer for this group to the message before the one under the current point, so the next time you read this group, the message indicated by the point will appear first.

Netnews Group Punt Messages (bound to **G** in News-Headers mode) [*Command*]

This command goes on to the next bulletin board in your group file. Without an argument, the system sets the pointer for the current group to the last message. With an argument, Hemlock sets the pointer to the last visible message in the group.

Netnews Exit (bound to **q** in News-Headers mode) [*Command*]
Netnews Exit Confirm (initial value **t**) [Hemlock *Variable*]

This command cleans up and deletes the News-Headers buffer and all associated News-Message buffers. If the value of Netnews Exit Confirm is **nil**, then Hemlock will not prompt before exiting.

## 11.5. Replying to Messages

The Hemlock Netnews interface also provides an easy way of replying to messages through the Hemlock Mailer or via Post mode.

Netnews Reply to Sender [*Command*]

When you invoke this command, Hemlock creates a Draft buffer and tries to fill in the *to* and *subject* fields of the draft. For the *to* field, Hemlock looks at the *reply-to* field of the message you are replying to, or failing that, the *from* field. If the *subject* field does not start with **Re:**, Hemlock inserts this string, signifying that this is a reply.

Netnews Reply to Sender in Other Window (bound to **r** in News-Headers and News-Message.) [*Command*]

This command splits the current window, placing the message you are replying to in the top window and a new Draft buffer in the bottom one. This command fills in the header fields in the same manner as Netnews Reply to Sender.

Netnews Reply to Group [*Command*]

This command creates a Post buffer with the *newsgroups* field set to the current group and the *subject* field constructed in the same way as in Netnews Reply to Sender.

Netnews Reply to Group in Other Window (bound to **R** in News-Headers and News-Message.) [*Command*]

This command splits the current window, placing the message you are replying to in the top window and a new Post buffer in the bottom one. This command will fill in the header fields in the same manner as Netnews Reply to Group.

Netnews Post Message (bound to **C-x P**) [*Command*]

This command creates a Post buffer. If you are in a News-Headers or News-Message buffer, Hemlock fills in the *newsgroups* field with the current group.

Netnews Forward Message (bound to **f** in News-Headers and News-Message modes.) [*Command*]

This command creates a Post buffer. If you are in a Netnews Headers or News-Message buffer, Hemlock will put the text of the current message into the buffer along with lines delimiting the forwarded message.

Netnews Goto Post Buffer (bound to **H-p** in News-Message mode) [*Command*]

This command, when invoked in a News-Message or Draft buffer with an associated News-Headers buffer, places the associated News-Headers buffer into the current window.

Netnews Goto Draft Buffer (bound to **H-d** in News-Message mode) [*Command*]

This command, when invoked in a News-Message buffer with an associated Draft buffer, places the Draft buffer into the current window.

## 11.6. Posting Messages

Netnews Deliver Post (bound to **H-s** in Post mode) [*Command*]
Netnews Deliver Post Confirm (initial value **t**) [Hemlock *Variable*]

This command delivers the contents of a Post buffer to the NNTP server. If Netnews Deliver Post Confirm is **t**, Hemlock will ask for confirmation before posting the message. Hemlock feeps you if NNTP does not accept the message.

Netnews Abort Post (bound to **H-q** in Post mode) [*Command*]

This command deletes the current Post buffer.

As in the mailer, when replying to a message you can be excerpt sections of it using Insert Message Buffer and Insert Message Region in Post and News-Message modes, respectively. You can also use these commands when replying to a message via mail in a Draft buffer. In all cases, the same binding is used: **H-y**.

## 11.7. Wallchart

**Global bindings:**

| | |
|---|---|
| Netnews Post Message | **C-x P** |

**News-Headers and News-Message modes bindings:**

| | |
|---|---|
| Netnews Next Article | **n** |
| Netnews Previous Article | **p** |
| Netnews Go to Next Group | **g** |
| Netnews Group Punt Messages | **G** |
| List All Groups | **l** |
| Netnews Append to File | **a** |
| Netnews Forward Message | **f** |
| Netnews Reply to Sender in Other Window | **r** |
| Netnews Reply to Group in Other Window | **R** |
| Netnews Quit Starting Here | **.** |

**News-Headers mode bindings:**

| | |
|---|---|
| Netnews Show Article | **Space** |
| Netnews Previous Line | **C-p**, **Uparrow** |
| Netnews Next Line | **C-n**, **Downarrow** |
| Netnews Headers Scroll Window Down | **C-v** |
| Netnews Headers Scroll Window Up | **M-v** |
| Netnews Select Message Buffer | **H-m** |
| Netnews Exit | **q** |
| Netnews Headers File Message | **o** |

**News-Message mode bindings:**

| | |
|---|---|
| Netnews Message Scroll Down | **Space** |
| Scroll Window Up | **Backspace** |
| Netnews Goto Headers Buffer | **H-h**, **^** |
| Netnews Message Keep Buffer | **k** |
| Netnews Message Quit | **q** |
| Netnews Message File Message | **o** |

| | |
|---|---|
| Netnews Goto Post Buffer | **H-p** |
| Netnews Goto Draft Buffer | **H-d** |
| Insert Message Region | **H-y** |

**Post mode bindings:**

| | |
|---|---|
| Netnews Select Message Buffer | **H-m** |
| Netnews Deliver Post | **H-s** |
| Netnews Abort Post | **H-q** |
| Insert Message Buffer | **H-y** |

**News-Browse mode bindings:**

| | |
|---|---|
| Netnews Quit Browse | **q** |
| Netnews Browse Add Group To File | **a** |
| Netnews Browse Read Group | **Space** |
| Next Line | **n** |
| Previous Line | **p** |

# Chapter 12

# System Interface

Hemlock provides a number of commands that access operating system resources such as the filesystem and print servers. These commands offer an alternative to leaving the editor and using the normal operating system command language (such as the Unix shell), but they are implementation dependent. Therefore, they might not even exist in some implementations.

## 12.1. File Utility Commands

This section describes some general file operation commands and quick directory commands.

See section 6.1 for a description Hemlock's directory editing mechanism, Dired mode.

Copy File [*Command*]

This command copies a file, allowing one wildcard in the filename. It prompts for source and destination specifications.

If these are both directories, then the copying process is recursive on the source, and if the destination is in the subdirectory structure of the source, the recursion excludes this portion of the directory tree. Use **`dir-spec-1/*`** to copy only the files in a source directory without recursively descending into subdirectories.

If the destination specification is a directory, and the source is a file, then it is copied into the destination with the same filename.

The copying process copies files maintaining the source's write date.

See the description of Dired Copy File Confirm, page 51, for controlling user interaction when the destination exists.

Rename File [*Command*]

This command renames a file, allowing one wildcard in the filename. It prompts for source and destination specifications.

If the destination is a directory, then the renaming process moves file(s) indicated by the source into the directory with their original filenames.

For Unix-based implementations, if you want to rename a directory, do not specify the trailing slash in the source specification.

Delete File [*Command*]

This command prompts for the name of a file and deletes it.

Directory (bound to **C-x C-d**) [*Command*]
Verbose Directory (bound to **C-x C-D**) [*Command*]

These commands prompt for a pathname (which may contain wildcards), and display a directory listing in a pop-up window. If a prefix argument is supplied, then normally hidden files such as Unix dot-files will also be displayed. Directory uses a compact, multiple-column format; Verbose Directory displays one file on a line, with information about protection, size, etc.

## 12.2. Printing

Print Region [*Command*]
Print Buffer [*Command*]
Print File [*Command*]

Print Region and Print Buffer print the contents of the current region and the current buffer, respectively. Print File prompts for a the name of a file and prints that file. Any error messages will be displayed in the echo area.

Print Utility (initial value **`"/usr/cs/bin/lpr"`**) [Hemlock *Variable*]
Print Utility Switches (initial value **`()`**) [Hemlock *Variable*]

Print Utility is the program the print commands use to send files to the printer. The program should act like **`lpr`**: if a filename is given as an argument, it should print that file, and if no name appears, standard input should be assumed. Print Utility Switches is a list of strings specifying the options to pass to the program.

## 12.3. Scribe

Scribe Buffer File (bound to **C-x `c`** in Scribe mode) [*Command*]
Scribe Buffer File Confirm (initial value **`t`**) [Hemlock *Variable*]
Scribe File [*Command*]

Scribe Buffer File invokes Scribe Utility on the file associated with the current buffer. That process's default directory is the directory of the file. The process sends its output to the Scribe Warnings buffer. Before doing anything, this asks the user to confirm saving and formatting the file. This prompting can be inhibited with "Scribe Buffer File Confirm".

Scribe File invokes Scribe Utility on a file supplied by the user in the same manner as describe above.

Scribe Utility (initial value **`"/usr/misc/bin/scribe"`**) [Hemlock *Variable*]
Scribe Utility Switches [Hemlock *Variable*]

Scribe Utility is the program the Scribe commands use to compile the text formatting. Scribe Utility Switches is a list of strings whose contents would be contiguous characters, other than space, had the user invoked this program on a command line outside of Hemlock. Do not include the name of the file to compile in this variable; the Scribe commands supply this.

Select Scribe Warnings (bound to **Scribe: C-M-C**) [*Command*]

This command makes the Scribe Warnings buffer current if it exists.

## 12.4. Miscellaneous

Manual Page [*Command*]

This command displays a Unix manual page in a buffer which is in View mode. When given an argument, this puts the manual page in a pop-up display.

Unix Filter Region [*Command*]

This command prompts for a UNIX program and then passes the current region to the program as standard input. The standard output from the program is used to replace the region. This command is undoable.

# Chapter 13

# 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.

## 13.1. 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.

Define Keyboard Macro Key (bound to **C-x M-(;** ) [*Command*]
Define Keyboard Macro Key Confirm (initial value `t`) [Hemlock *Variable*]

This command prompts for a key before going into a mode for defining keyboard macros. After defining the macro Hemlock binds it to the key. If the key is already bound, Hemlock asks for confirmation before clobbering the binding; this prompting can be inhibited by setting Define Keyboard Macro Key Confirm to **`nil`**.

Keyboard Macro Query (bound to **C-x q**) [*Command*]

This command conditionalizes the execution of a keyboard macro. When invoked during the definition of a macro, it does nothing. When the macro replays, it prompts the user for a key-event indicating what action to take. 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.

## 13.2. 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.

## 13.3. 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 variable and an expression, then sets the current value of the variable to the result of the evaluation of the expression.

Defhvar [*Command*]

Like Set Variable, this command prompts for the name of a Hemlock variable and an expression. Like Bind Key, this command prompts for a place: mode, buffer or local. The result of evaluating the expression is defined to be the value of the named variable in the specified place.

This command is most useful for making mode or buffer local bindings of variables. Redefining a variable in a mode or buffer will create a customization that takes effect only when in that mode or buffer.

Unlike Set Variable, the variable name need not be the name of an existing variable: new variables may be defined. If the variable is already defined in the current environment, Hemlock copies the documentation and hooks to the new definition.

## 13.4. Init Files

Hemlock customizations are normally put in Hemlock's initialization file, "**hemlock-init.lisp**", or when compiled "**hemlock-init.fasl**". When starting up Lisp, use the **-hinit** switch to indicate a particular file. 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 follows:

```
;;; -*- 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 string-search-ignore-case nil)
;;;
;;; Make "Query Replace" replace strings literally.
(setv case-replace nil)
```

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

# Index

# Index

# Table of Contents