Hemlock Command Implementor's Manual

Bill Chiles
Rob MacLachlan

February 1992

CMU-CS-89-134-R1

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is a revised version of Technical Report CMU-CS-87-159.

Abstract

This document describes how to write commands for the Hemlock text editor, as of 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 the 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.

Chapter 1

Introduction

Hemlock is a text editor which follows in the tradition of editors such as EMACS and the Lisp Machine editor ZWEI. In its basic form, Hemlock has almost the same command set as EMACS, and similar features such as multiple buffers and windows, extended commands, and built in documentation.

Both user extensions and the original commands are written in Lisp, therefore a command implementor will have a working knowledge of this language. Users not familiar with Lisp need not despair however. Many users of Multics EMACS, another text editor written in Lisp, came to learn Lisp simply for the purpose of writing their own editor extensions, and found, to their surprise, that it was really pretty easy to write simple commands.

This document describes the Common Lisp functions, macros and data structures that are used to implement new commands. The basic editor consists of a set of Lisp utility functions for manipulating buffers and the other data structures of the editor as well as handling the display. All user level commands are written in terms of these functions. To find out how to define commands see chapter 7.

Chapter 2

Representation of Text

2.1. Lines

In Hemlock all text is in some *line*. Text is broken into lines wherever it contains a newline character; newline characters are never stored, but are assumed to exist between every pair of lines. The implicit newline character is treated as a single character by the text primitives.

`linep` *line* [*Function*]

This function returns **`t`** if *line* is a **`line`** object, otherwise **`nil`**.

`line-string` *line* [*Function*]

Given a *line*, this function returns as a simple string the characters in the line. This is **`setf`**'able to set the **`line-string`** to any string that does not contain newline characters. It is an error to destructively modify the result of **`line-string`** or to destructively modify any string after the **`line-string`** of some line has been set to that string.

`line-previous` *line* [*Function*]
`line-next` *line* [*Function*]

Given a *line*, **`line-previous`** returns the previous line or **`nil`** if there is no previous line. Similarly, **`line-next`** returns the line following *line* or **`nil`**.

`line-buffer` *line* [*Function*]

This function returns the buffer which contains this *line*. Since a line may not be associated with any buffer, in which case **`line-buffer`** returns **`nil`**.

`line-length` *line* [*Function*]

This function returns the number of characters in the *line*. This excludes the newline character at the end.

`line-character` *line index* [*Function*]

This function returns the character at position *index* within *line*. It is an error for *index* to be greater than the length of the line or less than zero. If *index* is equal to the length of the line, this returns a **`#\newline`** character.

`line-plist` *line* [*Function*]

This function returns the property-list for *line*. **`setf`**, **`getf`**, **`putf`** and **`remf`** can be used to change properties. This is typically used in conjunction with **`line-signature`** to cache information about the line's contents.

line-signature *line* [*Function*]

This function returns an object that serves as a signature for a *line*'s contents. It is guaranteed that any modification of text on the line will result in the signature changing so that it is not **eql** to any previous value. The signature may change even when the text remains unmodified, but this does not happen often.

2.2. Marks

A mark indicates a specific position within the text represented by a line and a character position within that line. Although a mark is sometimes loosely referred to as pointing to some character, it in fact points between characters. If the **charpos** is zero, the previous character is the newline character separating the previous line from the mark's **line**. If the charpos is equal to the number of characters in the line, the next character is the newline character separating the current line from the next. If the mark's line has no previous line, a mark with **charpos** of zero has no previous character; if the mark's line has no next line, a mark with **charpos** equal to the length of the line has no next character.

This section discusses the very basic operations involving marks, but a lot of Hemlock programming is built on altering some text at a mark. For more extended uses of marks see chapter 4.

2.2.1. Kinds of Marks

A mark may have one of two lifetimes: *temporary* or *permanent*. Permanent marks remain valid after arbitrary operations on the text; temporary marks do not. Temporary marks are used because less bookkeeping overhead is involved in their creation and use. If a temporary mark is used after the text it points to has been modified results will be unpredictable. Permanent marks continue to point between the same two characters regardless of insertions and deletions made before or after them.

There are two different kinds of permanent marks which differ only in their behavior when text is inserted *at the position of the mark*; text is inserted to the left of a *left-inserting* mark and to the right of *right-inserting* mark.

2.2.2. Mark Functions

markp *mark* [*Function*]

This function returns **t** if *mark* is a **mark** object, otherwise **nil**.

mark-line *mark* [*Function*]

This function returns the line to which *mark* points.

mark-charpos *mark* [*Function*]

This function returns the character position of the character after *mark*. If *mark*'s line has no next line, this returns the length of the line as usual; however, there is actually is no character after the mark.

mark-kind *mark* [*Function*]

This function returns one of **:right-inserting**, **:left-inserting** or **:temporary** depending on the mark's kind. A corresponding **setf** form changes the mark's kind.

previous-character *mark* [*Function*]
next-character *mark* [*Function*]

This function returns the character immediately before (after) the position of the *mark*, or **nil** if there is no previous (next) character. These characters may be set with **setf** when they exist; the **setf** methods for these forms signal errors when there is no previous or next character.

2.2.3. Making Marks

`mark` *line charpos* **`&optional`** *kind* [*Function*]

This function returns a mark object that points to the *charpos*'th character of the *line*. *Kind* is the kind of mark to create, one of **`:temporary`**, **`:left-inserting`**, or **`:right-inserting`**. The default is **`:temporary`**.

`copy-mark` *mark* **`&optional`** *kind* [*Function*]

This function returns a new mark pointing to the same position and of the same kind, or of kind *kind* if it is supplied.

`delete-mark` *mark* [*Function*]

This function deletes *mark*. Delete any permanent marks when you are finished using it.

`with-mark` **`(`**{**`(`***mark pos* [*kind*]**`)`**}* **`)`** {*form*}* [*Macro*]

This macro binds to each variable *mark* a mark of kind *kind*, which defaults to **`:temporary`**, pointing to the same position as the mark *pos*. On exit from the scope the mark is deleted. The value of the last *form* is the value returned.

2.2.4. Moving Marks

These functions destructively modify marks to point to new positions. Other sections of this document describe mark moving routines specific to higher level text forms than characters and lines, such as words, sentences, paragraphs, Lisp forms, etc.

`move-to-position` *mark charpos* **`&optional`** *line* [*Function*]

This function changes the *mark* to point to the given character position on the line *line*. *Line* defaults to *mark*'s line.

`move-mark` *mark new-position* [*Function*]

This function moves *mark* to the same position as the mark *new-position* and returns it.

`line-start` *mark* **`&optional`** *line* [*Function*]
`line-end` *mark* **`&optional`** *line* [*Function*]

This function changes *mark* to point to the beginning or the end of *line* and returns it. *Line* defaults to *mark*'s line.

`buffer-start` *mark* **`&optional`** *buffer* [*Function*]
`buffer-end` *mark* **`&optional`** *buffer* [*Function*]

These functions change *mark* to point to the beginning or end of *buffer*, which defaults to the buffer *mark* currently points into. If *buffer* is unsupplied, then it is an error for *mark* to be disassociated from any buffer.

`mark-before` *mark* [*Function*]
`mark-after` *mark* [*Function*]

These functions change *mark* to point one character before or after the current position. If there is no character before/after the current position, then they return **`nil`** and leave *mark* unmodified.

character-offset *mark n* [*Function*]

This function changes *mark* to point *n* characters after (*n* before if *n* is negative) the current position. If there are less than *n* characters after (before) the *mark*, then this returns **nil** and *mark* is unmodified.

line-offset *mark n* **&optional** *charpos* [*Function*]

This function changes *mark* to point *n* lines after (*n* before if *n* is negative) the current position. The character position of the resulting mark is

(min (line-length *resulting-line***) (mark-charpos** *mark***))**

if *charpos* is unspecified, or

(min (line-length *resulting-line***)** *charpos***)**

if it is. As with `character-offset`, if there are not *n* lines then **nil** is returned and *mark* is not modified.

2.3. Regions

A region is simply a pair of marks: a starting mark and an ending mark. The text in a region consists of the characters following the starting mark and preceding the ending mark (keep in mind that a mark points between characters on a line, not at them).

By modifying the starting or ending mark in a region it is possible to produce regions with a start and end which are out of order or even in different buffers. The use of such regions is undefined and may result in arbitrarily bad behavior.

2.3.1. Region Functions

region *start end* [*Function*]

This function returns a region constructed from the marks *start* and *end*. It is an error for the marks to point to non-contiguous lines or for *start* to come after *end*.

regionp *region* [*Function*]

This function returns **t** if *region* is a **region** object, otherwise **nil**.

make-empty-region [*Function*]

This function returns a region with start and end marks pointing to the start of one empty line. The start mark is a **:right-inserting** mark, and the end is a **:left-inserting** mark.

copy-region *region* [*Function*]

This function returns a region containing a copy of the text in the specified *region*. The resulting region is completely disjoint from *region* with respect to data references — marks, lines, text, etc.

region-to-string *region* [*Function*]
string-to-region *string* [*Function*]

These functions coerce regions to Lisp strings and vice versa. Within the string, lines are delimited by newline characters.

line-to-region *line* [*Function*]

This function returns a region containing all the characters on *line*. The first mark is **:right-inserting** and the last is **:left-inserting**.

region-start *region* [*Function*]
region-end *region* [*Function*]

This function returns the start or end mark of *region*.

region-bounds *region* [*Function*]

This function returns as multiple-values the starting and ending marks of *region*.

set-region-bounds *region start end* [*Function*]

This function sets the start and end of region to *start* and *end*. It is an error for *start* to be after or in a different buffer from *end*.

count-lines *region* [*Function*]

This function returns the number of lines in the *region*, first and last lines inclusive. A newline is associated with the line it follows, thus a region containing some number of non-newline characters followed by one newline is one line, but if a newline were added at the beginning, it would be two lines.

count-characters *region* [*Function*]

This function returns the number of characters in a given *region*. This counts line breaks as one character.

check-region-query-size *region* [*Function*]
Region Query Size (initial value **30**) [Hemlock *Variable*]

check-region-query-size counts the lines in *region*, and if their number exceeds the Region Query Size threshold, it prompts the user for confirmation. This should be used in commands that perform destructive operations and are not undoable. If the user responds negatively, then this signals an editor-error, aborting whatever command was in progress.

Chapter 3

Buffers

A buffer is an environment within Hemlock consisting of:

1. A name.
2. A piece of text.
3. A current focus of attention, the point.
4. An associated file (optional).
5. A write protect flag.
6. Some variables (page 23).
7. Some key bindings (page 29).
8. Some collection of modes (page 35).
9. Some windows in which it is displayed (page 41).
10. A list of modeline fields (optional).

3.1. The Current Buffer

`current-buffer` [*Function*]
Set Buffer Hook [Hemlock *Variable*]
After Set Buffer Hook [Hemlock *Variable*]

`current-buffer` returns the current buffer object. Usually this is the buffer that **`current-window`** (page 41) is displaying. This value may be changed with **`setf`**, and the **`setf`** method invokes Set Buffer Hook before the change occurs with the new value. After the change occurs, the method invokes After Set Buffer Hook with the old value.

`current-point` [*Function*]

This function returns the **`buffer-point`** of the current buffer. This is such a common idiom in commands that it is defined despite its trivial implementation.

`current-mark` [*Function*]
`pop-buffer-mark` [*Function*]
`push-buffer-mark` *mark* **`&optional`** *activate-region* [*Function*]

`current-mark` returns the top of the current buffer's mark stack. There always is at least one mark at the beginning of the buffer's region, and all marks returned are right-inserting.

`pop-buffer-mark` pops the current buffer's mark stack, returning the mark. If the stack becomes empty, this pushes a new mark on the stack pointing to the buffer's start. This always deactivates the current region (see section 4.4).

push-buffer-mark pushes *mark* into the current buffer's mark stack, ensuring that the mark is right-inserting. If *mark* does not point into the current buffer, this signals an error. Optionally, the current region is made active, but this never deactivates the current region (see section 4.4). *Mark* is returned.

buffer-list [*Variable*]

This variable holds a list of all the buffer objects made with **make-buffer**.

buffer-names [*Variable*]

This variable holds a **string-table** (page 69) of all the names of the buffers in ***buffer-list***. The values of the entries are the corresponding buffer objects.

buffer-history [*Variable*]

This is a list of buffer objects ordered from those most recently selected to those selected farthest in the past. When someone makes a buffer, an element of Make Buffer Hook adds this buffer to the end of this list. When someone deletes a buffer, an element of Delete Buffer Hook removes the buffer from this list. Each buffer occurs in this list exactly once, but it never contains the ***echo-area-buffer***.

change-to-buffer *buffer* [*Function*]

This switches to *buffer* in the **current-window** maintaining **buffer-history**.

previous-buffer [*Function*]

This returns the first buffer from ***buffer-history*** that is not the **current-buffer**. If none can be found, then this returns **nil**.

3.2. Buffer Functions

make-buffer *name* **&key :modes :modeline-fields :delete-hook** [*Function*]
Make Buffer Hook [Hemlock *Variable*]
Default Modeline Fields [Hemlock *Variable*]

make-buffer creates and returns a buffer with the given *name*. If a buffer named *name* already exists, **nil** is returned. *Modes* is a list of modes which should be in effect in the buffer, major mode first, followed by any minor modes. If this is omitted then the buffer is created with the list of modes contained in Default Modes (page 35). *Modeline-fields* is a list of modeline-field objects (see section 3.3) which may be **nil**. **delete-hook** is a list of delete hooks specific to this buffer, and **delete-buffer** invokes these along with Delete Buffer Hook.

Buffers created with **make-buffer** are entered into the list ***buffer-list***, and their names are inserted into the string-table ***buffer-names***. When a buffer is created the hook Make Buffer Hook is invoked with the new buffer.

bufferp *buffer* [*Function*]

Returns **t** if *buffer* is a **buffer** object, otherwise **nil**.

buffer-name *buffer* [*Function*]
Buffer Name Hook [Hemlock *Variable*]

buffer-name returns the name, which is a string, of the given *buffer*. The corresponding **setf** method invokes Buffer Name Hook with *buffer* and the new name and then sets the buffer's name. When the user supplies a name for which a buffer already exists, the **setf** method signals an error.

buffer-region *buffer* [*Function*]

Returns the *buffer*'s region. This can be set with **setf**. Note, this returns the region that contains all the text in a buffer, not the **current-region** (page 19).

buffer-pathname *buffer* [*Function*]
Buffer Pathname Hook [Hemlock *Variable*]

buffer-pathname returns the pathname of the file associated with the given *buffer*, or nil if it has no associated file. This is the truename of the file as of the most recent time it was read or written. There is a **setf** form to change the pathname. When the pathname is changed the hook Buffer Pathname Hook is invoked with the buffer and new value.

buffer-write-date *buffer* [*Function*]

Returns the write date for the file associated with the buffer in universal time format. When this the **buffer-pathname** is set, use **setf** to set this to the corresponding write date, or to **nil** if the date is unknown or there is no file.

buffer-point *buffer* [*Function*]

Returns the mark which is the current location within *buffer*. To move the point, use **move-mark** or **move-to-position** (page 5) rather than setting **buffer-point** with **setf**.

buffer-mark *buffer* [*Function*]

This function returns the top of *buffer*'s mark stack. There always is at least one mark at the beginning of *buffer*'s region, and all marks returned are right-inserting.

buffer-start-mark *buffer* [*Function*]
buffer-end-mark *buffer* [*Function*]

These functions return the start and end marks of *buffer*'s region:

```
(buffer-start-mark buffer)  <==>
  (region-start (buffer-region buffer))
and
(buffer-end-mark buffer)  <==>
  (region-end (buffer-region buffer))
```

buffer-writable *buffer* [*Function*]
Buffer Writable Hook [Hemlock *Variable*]

This function returns **t** if you can modify the *buffer*, **nil** if you cannot. If a buffer is not writable, then any attempt to alter text in the buffer results in an error. There is a **setf** method to change this value.

The **setf** method invokes the functions in Buffer Writable Hook on the buffer and new value before storing the new value.

`buffer-modified` *buffer* [*Function*]
Buffer Modified Hook [Hemlock *Variable*]

> **`buffer-modified`** returns **`t`** if the *buffer* has been modified, **`nil`** if it hasn't. This attribute is set whenever a text-altering operation is performed on a buffer. There is a **`setf`** method to change this value.
>
> The **`setf`** method invokes the functions in Buffer Modified Hook with the buffer whenever the value of the modified flag changes.

`with-writable-buffer (`*buffer***`) &rest`** *forms* [*Macro*]

> This macro executes *forms* with *buffer*'s writable status set. After *forms* execute, this resets the *buffer*'s writable and modified status.

`buffer-signature` *buffer* [*Function*]

> This function returns an arbitrary number which reflects the buffer's current *signature*. The result is **`eql`** to a previous result if and only if the buffer has not been modified between the calls.

`buffer-variables` *buffer* [*Function*]

> This function returns a string-table (page 69) containing the names of the buffer's local variables. See chapter 6.

`buffer-modes` *buffer* [*Function*]

> This function returns the list of the names of the modes active in *buffer*. The major mode is first, followed by any minor modes. See chapter 8.

`buffer-windows` *buffer* [*Function*]

> This function returns the list of all the windows in which the buffer may be displayed. This list may include windows which are not currently visible. See page 41 for a discussion of windows.

`buffer-delete-hook` *buffer* [*Function*]

> This function returns the list of buffer specific functions **`delete-buffer`** invokes when deleting a buffer. This is **`setf`**'able.

`delete-buffer` *buffer* [*Function*]
Delete Buffer Hook [Hemlock *Variable*]

> **`delete-buffer`** removes *buffer* from **`*buffer-list*`** (page 10) and its name from **`*buffer-names*`** (page 10). Before *buffer* is deleted, this invokes the functions on *buffer* returned by **`buffer-delete-hook`** and those found in Delete Buffer Hook. If *buffer* is the **`current-buffer`**, or if it is displayed in any windows, then this function signals an error.

`delete-buffer-if-possible` *buffer* [*Function*]

> This uses **`delete-buffer`** to delete *buffer* if at all possible. If *buffer* is the **`current-buffer`**, then this sets the **`current-buffer`** to the first distinct buffer in **`buffer-history`**. If *buffer* is displayed in any windows, then this makes each window display the same distinct buffer.

3.3. Modelines

A Buffer may specify a modeline, a line of text which is displayed across the bottom of a window to indicate status information. Modelines are described as a list of **`modeline-field`** objects which have individual update functions and are optionally fixed-width. These have an **eql** name for convenience in referencing and updating, but the name must be unique for all created modeline-field objects. When creating a modeline-field with a specified width, the result of the update function is either truncated or padded on the right to meet the constraint. All modeline-field functions must return simple strings with standard characters, and these take a buffer and a window as arguments. Modeline-field objects are typically shared amongst, or aliased by, different buffers' modeline fields lists. These lists are unique allowing fields to behave the same wherever they occur, but different buffers may display these fields in different arrangements.

Whenever one of the following changes occurs, all of a buffer's modeline fields are updated:

- A buffer's major mode is set.
- One of a buffer's minor modes is turned on or off.
- A buffer is renamed.
- A buffer's pathname changes.
- A buffer's modified status changes.
- A window's buffer is changed.

The policy is that whenever one of these changes occurs, it is guaranteed that the modeline will be updated before the next trip through redisplay. Furthermore, since the system cannot know what modeline-field objects the user has added whose update functions rely on these values, or how he has changed Default Modeline Fields, we must update all the fields. When any but the last occurs, the modeline-field update function is invoked once for each window into the buffer. When a window's buffer changes, each modeline-field update function is invoked once; other windows' modeline fields should not be affected due to a given window's buffer changing.

The user should note that modelines can be updated at any time, so update functions should be careful to avoid needless delays (for example, waiting for a local area network to determine information).

make-modeline-field &key :name :width :function [*Function*]
modeline-field-p *modeline-field* [*Function*]
modeline-field-name *modeline-field* [*Function*]

> **make-modeline-field** returns a modeline-field object with *name*, *width*, and *function*. *Width* defaults to **nil** meaning that the field is variable width; otherwise, the programmer must supply this as a positive integer. *Function* must take a buffer and window as arguments and return a **simple-string** containing only standard characters. If *name* already names a modeline-field object, then this signals an error.
>
> **modeline-field-name** returns the name field of a modeline-field object. If this is set with **setf**, and the new name already names a modeline-field, then the **setf** method signals an error.
>
> **modeline-field-p** returns **t** or **nil**, depending on whether its argument is a **modeline-field** object.

modeline-field *name* [*Function*]

> This returns the modeline-field object named *name*. If none exists, this returns nil.

modeline-field-function *modeline-field* [*Function*]

> Returns the function called when updating the *modeline-field*. When this is set with **setf**, the **setf** method updates *modeline-field* for all windows on all buffers that contain the given field, so the next trip through redisplay will reflect the change. All modeline-field functions must return simple strings with standard characters, and they take a buffer and a window as arguments.

`modeline-field-width` *modeline-field* [*Function*]

Returns the width to which *modeline-field* is constrained, or **`nil`** indicating that it is variable width. When this is set with **`setf`**, the **`setf`** method updates all modeline-fields for all windows on all buffers that contain the given field, so the next trip through redisplay will reflect the change. All the fields for any such modeline display must be updated, which is not the case when setting a modeline-field's function.

`buffer-modeline-fields` *buffer* [*Function*]

Returns a copy of the list of *buffer*'s modeline-field objects. This list can be destructively modified without affecting display of *buffer*'s modeline, but modifying any particular field's components (for example, width or function) causes the changes to be reflected the next trip through redisplay in every modeline display that uses the modified modeline-field. When this is set with **`setf`**, **`update-modeline-fields`** is called for each window into *buffer*.

`buffer-modeline-field-p` *buffer field* [*Function*]

If *field*, a modeline-field or the name of one, is in buffer's list of modeline-field objects, it is returned; otherwise, this returns nil.

`update-modeline-fields` *buffer window* [*Function*]

This invokes each modeline-field object's function from *buffer*'s list, passing *buffer* and *window*. The results are collected regarding each modeline-field object's width as appropriate, and the window is marked so the next trip through redisplay will reflect the changes. If window does not display modelines, then no computation occurs.

`update-modeline-field` *buffer window field-or-name* **[***Function*

This invokes the modeline-field object's function for *field-or-name*, which is a modeline-field object or the name of one for *buffer*. This passes *buffer* and *window* to the update function. The result is applied to the *window*'s modeline display using the modeline-field object's width, and the window is marked so the next trip through redisplay will reflect the changes. If the window does not display modelines, then no computation occurs. If *field-or-name* is not found in *buffer*'s list of modeline-field objects, then this signals an error. See **`buffer-modeline-field-p`** above.

Chapter 4

Altering and Searching Text

4.1. Altering Text

A note on marks and text alteration: **:temporary** marks are invalid after any change has been made to the text the mark points to; it is an error to use a temporary mark after such a change has been made. If text is deleted which has permanent marks pointing into it then they are left pointing to the position where the text was.

insert-character *mark character* [*Function*]
insert-string *mark string* [*Function*]
insert-region *mark region* [*Function*]

Inserts *character*, *string* or *region* at *mark*. **insert-character** signals an error if *character* is not **string-char-p**. If *string* or *region* is empty, and *mark* is in some buffer, then Hemlock leaves **buffer-modified** of *mark*'s buffer unaffected.

ninsert-region *mark region* [*Function*]

Like **insert-region**, inserts the *region* at the *mark*'s position, destroying the source region. This must be used with caution, since if anyone else can refer to the source region bad things will happen. In particular, one should make sure the region is not linked into any existing buffer. If *region* is empty, and *mark* is in some buffer, then Hemlock leaves **buffer-modified** of *mark*'s buffer unaffected.

delete-characters *mark n* [*Function*]

This deletes *n* characters after the *mark* (or -*n* before if *n* is negative). If *n* characters after (or -*n* before) the *mark* do not exist, then this returns **nil**; otherwise, it returns **t**. If *n* is zero, and *mark* is in some buffer, then Hemlock leaves **buffer-modified** of *mark*'s buffer unaffected.

delete-region *region* [*Function*]

This deletes *region*. This is faster than **delete-and-save-region** (below) because no lines are copied. If *region* is empty and contained in some buffer's **buffer-region**, then Hemlock leaves **buffer-modified** of the buffer unaffected.

delete-and-save-region *region* [*Function*]

This deletes *region* and returns a region containing the original *region*'s text. If *region* is empty and contained in some buffer's **buffer-region**, then Hemlock leaves **buffer-modified** of the buffer unaffected. In this case, this returns a distinct empty region.

`filter-region` *function region* [*Function*]

Destructively modifies *region* by replacing the text of each line with the result of the application of *function* to a string containing that text. *Function* must obey the following restrictions:

1. The argument may not be destructively modified.
2. The return value may not contain newline characters.
3. The return value may not be destructively modified after it is returned from *function*.

The strings are passed in order, and are always simple strings.

Using this function, a region could be uppercased by doing:

```
(filter-region #'string-upcase region)
```

4.2. Text Predicates

`start-line-p` *mark* [*Function*]

Returns **`t`** if the *mark* points before the first character in a line, **`nil`** otherwise.

`end-line-p` *mark* [*Function*]

Returns **`t`** if the *mark* points after the last character in a line and before the newline, **`nil`** otherwise.

`empty-line-p` *mark* [*Function*]

Return **`t`** of the line which *mark* points to contains no characters.

`blank-line-p` *line* [*Function*]

Returns **`t`** if *line* contains only characters with a Whitespace attribute of 1. See chapter 9 for discussion of character attributes.

`blank-before-p` *mark* [*Function*]
`blank-after-p` *mark* [*Function*]

These functions test if all the characters preceding or following *mark* on the line it is on have a Whitespace attribute of **`1`**.

`same-line-p` *mark1 mark2* [*Function*]

Returns **`t`** if *mark1* and *mark2* point to the same line, or **`nil`** otherwise; That is,

```
(same-line-p a b) <==> (eq (mark-line a) (mark-line b))
```

`mark<` *mark1 mark2* [*Function*]
`mark<=` *mark1 mark2* [*Function*]
`mark=` *mark1 mark2* [*Function*]
`mark/=` *mark1 mark2* [*Function*]
`mark>=` *mark1 mark2* [*Function*]
`mark>` *mark1 mark2* [*Function*]

These predicates test the relative ordering of two marks in a piece of text, that is a mark is **`mark>`** another if it points to a position after it. If the marks point into different, non-connected pieces of text, such as different buffers, then it is an error to test their ordering; for such marks **`mark=`** is always false and **`mark/=`** is always true.

line< *line1 line2* [*Function*]
line<= *line1 line2* [*Function*]
line>= *line1 line2* [*Function*]
line> *line1 line2* [*Function*]

These predicates test the ordering of *line1* and *line2*. If the lines are in unconnected pieces of text it is an error to test their ordering.

lines-related *line1 line2* [*Function*]

This function returns **t** if *line1* and *line2* are in the same piece of text, or **nil** otherwise.

first-line-p *mark* [*Function*]
last-line-p *mark* [*Function*]

first-line-p returns **t** if there is no line before the line *mark* is on, and **nil** otherwise. *Last-line-p* similarly tests tests whether there is no line after *mark*.

4.3. Kill Ring

kill-ring [*Variable*]

This is a ring (see section 16.2) of regions deleted from buffers. Some commands save affected regions on the kill ring before performing modifications. You should consider making the command undoable (see section 16.3), but this is a simple way of achieving a less satisfactory means for the user to recover.

kill-region *region current-type* [*Function*]

This kills *region* saving it in ***kill-ring***. *Current-type* is either **:kill-forward** or **:kill-backward**. When the **last-command-type** (page 31) is one of these, this adds *region* to the beginning or end, respectively, of the top of ***kill-ring***. The result of calling this is undoable using the command Undo (see the *Hemlock User's Manual*). This sets **last-command-type** to *current-type*, and it interacts with **kill-characters**.

kill-characters *mark count* [*Function*]
Character Deletion Threshold (initial value **5**) [Hemlock *Variable*]

kill-characters kills *count* characters after *mark* if *count* is positive, otherwise before *mark* if *count* is negative. When *count* is greater than or equal to Character Deletion Threshold, the killed characters are saved on ***kill-ring***. This may be called multiple times contiguously (that is, without **last-command-type** (page 31) being set) to accumulate an effective count for purposes of comparison with the threshold.

This sets **last-command-type**, and it interacts with **kill-region**. When this adds a new region to ***kill-ring***, it sets **last-command-type** to **:kill-forward** (if *count* is positive) or **:kill-backward** (if *count* is negative). When **last-command-type** is **:kill-forward** or **:kill-backward**, this adds the killed characters to the beginning (if *count* is negative) or the end (if *count* is positive) of the top of ***kill-ring***, and it sets **last-command-type** as if it added a new region to ***kill-ring***. When the kill ring is unaffected, this sets **last-command-type** to **:char-kill-forward** or **:char-kill-backward** depending on whether *count* is positive or negative, respectively.

This returns mark if it deletes characters. If there are not *count* characters in the appropriate direction, this returns nil.

4.4. Active Regions

Every buffer has a mark stack (page 9) and a mark known as the point where most text altering nominally occurs. Between the top of the mark stack, the **`current-mark`**, and the **`current-buffer`**'s point, the **`current-point`**, is what is known as the **`current-region`**. Certain commands signal errors when the user tries to operate on the **`current-region`** without its having been activated. If the user turns off this feature, then the **`current-region`** is effectively always active.

When writing a command that marks a region of text, the programmer should make sure to activate the region. This typically occurs naturally from the primitives that you use to mark regions, but sometimes you must explicitly activate the region. These commands should be written this way, so they do not require the user to separately mark an area and then activate it. Commands that modify regions do not have to worry about deactivating the region since modifying a buffer automatically deactivates the region. Commands that insert text often activate the region ephemerally; that is, the region is active for the immediately following command, allowing the user wants to delete the region inserted, fill it, or whatever.

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**.

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

When this variable is non-**`nil`**, some primitives signal an editor-error if the region is not active. This may be set to **`nil`** for more traditional Emacs region semantics.

`*ephemerally-active-command-types*` [*Variable*]

This is a list of command types (see section 7.3), and its initial value is the list of **`:ephemerally-active`** and **`:unkill`**. When the previous command's type is one of these, the **`current-region`** is active for the currently executing command only, regardless of whether it does something to deactivate the region. However, the current command may activate the region for future commands. **`:ephemerally-active`** is a default command type that may be used to ephemerally activate the region, and **`:unkill`** is the type used by two commands, Un-kill and Rotate Kill Ring (what users typically think of as **C-y** and **M-y**).

`activate-region` [*Function*]

This makes the **`current-region`** active.

`deactivate-region` [*Function*]

After invoking this the **`current-region`** is no longer active.

`region-active-p` [*Function*]

Returns whether the **`current-region`** is active, including ephemerally. This ignores Active Regions Enabled.

check-region-active [*Function*]

This signals an editor-error when active regions are enabled, and the **current-region** is not active.

current-region &optional *error-if-not-active deactivate-region* [*Function*]

This returns a region formed with **current-mark** and **current-point**, optionally signaling an editor-error if the current region is not active. *Error-if-not-active* defaults to **t**. Each call returns a distinct region object. Depending on *deactivate-region* (defaults to **t**), fetching the current region deactivates it. Hemlock primitives are free to modify text regardless of whether the region is active, so a command that checks for this can deactivate the region whenever it is convenient.

4.5. Searching and Replacing

Before using any of these functions to do a character search, look at character attributes (page 37). They provide a facility similar to the syntax table in real EMACS. Syntax tables are a powerful, general, and efficient mechanism for assigning meanings to characters in various modes.

search-char-code-limit [*Constant*]

An exclusive upper limit for the char-code of characters given to the searching functions. The result of searches for characters with a char-code greater than or equal to this limit is ill-defined, but it is *not* an error to do such searches.

new-search-pattern *kind direction pattern* **&optional** *result-search-pattern* [*Function*]

Returns a *search-pattern* object which can be given to the **find-pattern** and **replace-pattern** functions. A search-pattern is a specification of a particular sort of search to do. *direction* is either **:forward** or **:backward**, indicating the direction to search in. *kind* specifies the kind of search pattern to make, and *pattern* is a thing which specifies what to search for.

The interpretation of *pattern* depends on the *kind* of pattern being made. Currently defined kinds of search pattern are:

:string-insensitive
: Does a case-insensitive string search, *pattern* being the string to search for.

:string-sensitive
: Does a case-sensitive string search for *pattern*.

:character
: Finds an occurrence of the character *pattern*. This is case sensitive.

:not-character
: Find a character which is not the character *pattern*.

:test
: Finds a character which satisfies the function *pattern*. This function may not be applied an any particular fashion, so it should depend only on what its argument is, and should have no side-effects.

:test-not
: Similar to as **:test**, except it finds a character that fails the test.

:any
: Finds a character that is in the string *pattern*.

:not-any
: Finds a character that is not in the string *pattern*.

result-search-pattern, if supplied, is a search-pattern to destructively modify to produce the new pattern. Where reasonable this should be supplied, since some kinds of search patterns may involve large data structures.

search-pattern-p *search-pattern* [*Function*]

Returns **t** if *search-pattern* is a **search-pattern** object, otherwise **nil**.

get-search-pattern *string direction* [*Function*]
last-search-pattern [*Variable*]
last-search-string [*Variable*]

get-search-pattern interfaces to a default search string and pattern that search and replacing commands can use. These commands then share a default when prompting for what to search or replace, and save on consing a search pattern each time they execute. This uses Default Search Kind (see the *Hemlock User's Manual*) when updating the pattern object. This returns the pattern, so you probably don't need to refer to ***last-search-pattern***, but ***last-search-string*** is useful when prompting.

find-pattern *mark search-pattern* [*Function*]

Find the next match of *search-pattern* starting at *mark*. If a match is found then *mark* is altered to point before the matched text and the number of characters matched is returned. If no match is found then **nil** is returned and *mark* is not modified.

replace-pattern *mark search-pattern replacement* **&optional** *n* [*Function*]

Replace *n* matches of *search-pattern* with the string *replacement* starting at *mark*. If *n* is **nil** (the default) then replace all matches. A mark pointing before the last replacement done is returned.

Chapter 5

The Current Environment

5.1. Different Scopes

In Hemlock the values of *variables* (page 23), *key-bindings* (page 29) and *character-attributes* (page 37) may depend on the **`current-buffer`** (page 9) and the modes active in it. There are three possible scopes for Hemlock values:

buffer local	The value is present only if the buffer it is local to is the **`current-buffer`**.
mode local	The value is present only when the mode it is local to is active in the **`current-buffer`**.
global	The value is always present unless shadowed by a buffer or mode local value.

5.2. Shadowing

It is possible for there to be a conflict between different values for the same thing in different scopes. For example, there be might a global binding for a given variable and also a local binding in the current buffer. Whenever there is a conflict shadowing occurs, permitting only one of the values to be visible in the current environment.

The process of resolving such a conflict can be described as a search down a list of places where the value might be defined, returning the first value found. The order for the search is as follows:

1. Local values in the current buffer.
2. Mode local values in the minor modes of the current buffer, in order from the highest precedence mode to the lowest precedence mode. The order of minor modes with equal precedences is undefined.
3. Mode local values in the current buffer's major mode.
4. Global values.

Chapter 6

Hemlock Variables

Hemlock implements a system of variables separate from normal Lisp variables for the following reasons:

1. Hemlock has different scoping rules which are useful in an editor. Hemlock variables can be local to a *buffer* (page 9) or a *mode* (page 35).
2. Hemlock variables have *hooks* (page 25), lists of functions called when someone sets the variable. See **`variable-value`** for the arguments Hemlock passes to these hook functions.
3. There is a database of variable names and documentation which makes it easier to find out what variables exist and what their values mean.

6.1. Variable Names

To the user, a variable name is a case insensitive string. This string is referred to as the *string name* of the variable. A string name is conventionally composed of words separated by spaces.

In Lisp code a variable name is a symbol. The name of this symbol is created by replacing any spaces in the string name with hyphens. This symbol name is always interned in the Hemlock package and referring to a symbol with the same name in the wrong package is an error.

`*global-variable-names*` [*Variable*]

This variable holds a string-table of the names of all the global Hemlock variables. The value of each entry is the symbol name of the variable.

`current-variable-tables` [*Function*]

This function returns a list of variable tables currently established, globally, in the **`current-buffer`**, and by the modes of the **`current-buffer`**. This list is suitable for use with **`prompt-for-variable`**.

6.2. Variable Functions

In the following descriptions *name* is the symbol name of the variable.

`defhvar` *string-name documentation* **`&key :mode :buffer :hooks :value`** [*Function*]

This function defines a Hemlock variable. Functions that take a variable name signal an error when the variable is undefined.

string-name The string name of the variable to define.

documentation The documentation string for the variable.

:mode, **:buffer** If *buffer* is supplied, the variable is local to that buffer. If *mode* is supplied, it is local to that mode. If neither is supplied, it is global.

:value This is the initial value for the variable, which defaults to **nil**.

:hooks This is the initial list of functions to call when someone sets the variable's value. These functions execute before Hemlock establishes the new value. See **variable-value** for the arguments passed to the hook functions.

If a variable with the same name already exists in the same place, then **defhvar** sets its hooks and value from *hooks* and *value* if the user supplies these keywords.

variable-value *name* **&optional** *kind where* [*Function*]

This function returns the value of a Hemlock variable in some place. The following values for *kind* are defined:

:current Return the value present in the current environment, taking into consideration any mode or buffer local variables. This is the default.

:global Return the global value.

:mode Return the value in the mode named *where*.

:buffer Return the value in the buffer *where*.

When set with **setf**, Hemlock sets the value of the specified variable and invokes the functions in its hook list with *name*, *kind*, *where*, and the new value.

variable-documentation *name* **&optional** *kind where* [*Function*]
variable-hooks *name* **&optional** *kind where* [*Function*]
variable-name *name* **&optional** *kind where* [*Function*]

These function return the documentation, hooks and string name of a Hemlock variable. The *kind* and *where* arguments are the same as for **variable-value**. The documentation and hook list may be set using **setf**.

string-to-variable *string* [*Function*]

This function converts a string into the corresponding variable symbol name. *String* need not be the name of an actual Hemlock variable.

value *name* [*Macro*]
setv *name new-value* [*Macro*]

These macros get and set the current value of the Hemlock variable *name*. *Name* is not evaluated. There is a **setf** form for **value**.

hlet ({**(***var value***)**}***)** {*form*}* [*Macro*]

This macro is very similar to **let** in effect; within its scope each of the Hemlock variables *var* have the respective *value*s, but after the scope is exited by any means the binding is removed. This does not cause any hooks to be invoked. The value of the last *form* is returned.

hemlock-bound-p *name* **&optional** *kind where* [*Function*]

Returns **t** if *name* is defined as a Hemlock variable in the place specified by *kind* and *where*, or **nil** otherwise.

delete-variable *name* **&optional** *kind where* [*Function*]
Delete Variable Hook [Hemlock *Variable*]

delete-variable makes the Hemlock variable *name* no longer defined in the specified place. *Kind* and *where* have the same meanings as they do for **variable-value**, except that **:current** is not available, and the default for *kind* is **:global**

An error will be signaled if no such variable exists. The hook, Delete Variable Hook is invoked with the same arguments before the variable is deleted.

6.3. Hooks

Hemlock actions such as setting variables, changing buffers, changing windows, turning modes on and off, etc., often have hooks associated with them. A hook is a list of functions called before the system performs the action. The manual describes the object specific hooks with the rest of the operations defined on these objects.

Often hooks are stored in Hemlock variables, Delete Buffer Hook and Set Window Hook for example. This leads to a minor point of confusion because these variables have hooks that the system executes when someone changes their values. These hook functions Hemlock invokes when someone sets a variable are an example of a hook stored in an object instead of a Hemlock variable. These are all hooks for editor activity, but Hemlock keeps them in different kinds of locations. This is why some of the routines in this section have a special interpretation of the hook *place* argument.

add-hook *place hook-fun* [*Macro*]
remove-hook *place hook-fun* [*Macro*]

These macros add or remove a hook function in some *place*. If *hook-fun* already exists in *place*, this call has no effect. If *place* is a symbol, then it is a Hemlock variable; otherwise, it is a generalized variable or storage location. Here are two examples:

```
(add-hook delete-buffer-hook 'remove-buffer-from-menu)

(add-hook (variable-hooks 'check-mail-interval)
          'reschedule-mail-check)
```

invoke-hook *place* **&rest** *args* [*Macro*]

This macro calls all the functions in *place*. If *place* is a symbol, then it is a Hemlock variable; otherwise, it is a generalized variable.

Chapter 7

Commands

7.1. Introduction

The way that the user tells Hemlock to do something is by invoking a *command*. Commands have three attributes:

name	A command's name provides a way to refer to it. Command names are usually capitalized words separated by spaces, such as Forward Word.
documentation	The documentation for a command is used by on-line help facilities.
function	A command is implemented by a Lisp function, which is callable from Lisp.

`*command-names*` [*Variable*]

Holds a string-table (page 69) associating command names to command objects. Whenever a new command is defined it is entered in this table.

7.1.1. Defining Commands

`defcommand` {*command-name* | **(***command-name function-name***)**} *lambda-list* *command-doc function-doc* {*form*}* [*Macro*]

Defines a command named *name*. **`defcommand`** creates a function to implement the command from the *lambda-list* and *form*'s supplied. The *lambda-list* must specify one required argument, see section 7.4, which by convention is typically named **`p`**. If the caller does not specify *function-name*, **`defcommand`** creates the command name by replacing all spaces with hyphens and appending "**`-command`**". *Function-doc* becomes the documentation for the function and should primarily describe issues involved in calling the command as a function, such as what any additional arguments are. *Command-doc* becomes the command documentation for the command.

`make-command` *name documentation function* [*Function*]

Defines a new command named *name*, with command documentation *documentation* and function *function*. The command in entered in the string-table **`*command-names*`** (page 27), with the command object as its value. Normally command implementors will use the **`defcommand`** macro, but this permits access to the command definition mechanism at a lower level, which is occasionally useful.

`commandp` *command* [*Function*]

Returns **`t`** if *command* is a **`command`** object, otherwise **`nil`**.

command-documentation *command* [*Function*]
command-function *command* [*Function*]
command-name *command* [*Function*]

Returns the documentation, function, or name for *command*. These may be set with **setf**.

7.1.2. Command Documentation

Command documentation is a description of what the command does when it is invoked as an extended command or from a key. Command documentation may be either a string or a function. If the documentation is a string then the first line should briefly summarize the command, with remaining lines filling the details. Example:

```
(defcommand "Forward Character" (p)
  "Move the point forward one character.
   With prefix argument move that many characters, with negative
   argument go backwards."
  "Move the point of the current buffer forward p characters."
   . . .)
```

Command documentation may also be a function of one argument. The function is called with either **:short** or **:full**, indicating that the function should return a short documentation string or do something to document the command fully.

7.2. The Command Interpreter

The *command interpreter* is a function which reads key-events (see section 7.2.1) from the keyboard and dispatches to different commands on the basis of what the user types. When the command interpreter executes a command, we say it *invokes* the command. The command interpreter also provides facilities for communication between commands contiguously running commands, such as a last command type register. It also takes care of resetting communication mechanisms, clearing the echo area, displaying partial keys typed slowly by the user, etc.

invoke-hook [*Variable*]

This variable contains a function the command interpreter calls when it wants to invoke a command. The function receives the command and the prefix argument as arguments. The initial value is a function which simply funcalls the **command-function** of the command with the supplied prefix argument. This is useful for implementing keyboard macros and similar things.

Command Abort Hook [Hemlock *Variable*]

The command interpreter invokes the function in this variable whenever someone aborts a command (for example, if someone called **editor-error**).

When Hemlock initially starts the command interpreter is in control, but commands may read from the keyboard themselves and assign whatever interpretation they will to the key-events read. Commands may call the command interpreter recursively using the function **recursive-edit** (page 32).

7.2.1. Editor Input

The canonical representation of editor input is a key-event structure. Users can bind commands to keys (see section 7.2.2), which are non-zero length sequences of key-events. A key-event consists of an identifying token known as a *keysym* and a field of bits representing modifiers. Users define keysyms, integers between 0 and 65535 inclusively, by supplying names that reflect the legends on their keyboard's keys. Users define modifier names similarly, but the system chooses the bit and mask for recognizing the modifier. You can use keysym and modifier names to textually specify key-events and Hemlock keys in a **#k** syntax. The following are some examples:

```
#k"C-u"
#k"Control-u"
#k"c-m-z"
#k"control-x meta-d"
#k"a"
#k"A"
#k"Linefeed"
```

This is convenient for use within code and in init files containing **bind-key** calls.

The **#k** syntax is delimited by double quotes, but the system parses the contents rather than reading it as a Common Lisp string. Within the double quotes, spaces separate multiple key-events. A single key-event optionally starts with modifier names terminated by hyphens. Modifier names are alphabetic sequences of characters which the system uses case-insensitively. Following modifiers is a keysym name, which is case-insensitive if it consists of multiple characters, but if the name consists of only a single character, then it is case-sensitive.

You can escape special characters — hyphen, double quote, open angle bracket, close angle bracket, and space — with a backslash, and you can specify a backslash by using two contiguously. You can use angle brackets to enclose a keysym name with many special characters in it. Between angle brackets appearing in a keysym name position, there are only two special characters, the closing angle bracket and backslash.

For more information on key-events see section 18.1.

7.2.2. Binding Commands to Keys

The command interpreter determines which command to invoke on the basis of *key bindings*. A key binding is an association between a command and a sequence of key-events (see section 7.2.1. A sequence of key-events is called a *key* and is represented by a single key-event or a sequence (list or vector) of key-events.

Since key bindings may be local to a mode or buffer, the current environment (page 21) determines the set of key bindings in effect at any given time. When the command interpreter tries to find the binding for a key, it first checks if there is a local binding in the **current-buffer** (page 9), then if there is a binding in each of the minor modes and the major mode for the current buffer (page 35), and finally checks to see if there is a global binding. If no binding is found, then the command interpreter beeps or flashes the screen to indicate this.

bind-key *name key* **&optional** *kind where* [*Function*]

This function associates command *name* and *key* in some environment. *Key* is either a key-event or a sequence of key-events. There are three possible values of *kind*:

- **:global** The default, make a global key binding.
- **:mode** Make a mode specific key binding in the mode whose name is *where*.
- **:buffer** Make a binding which is local to buffer *where*.

This processes *key* for key translations before establishing the binding. See section 7.2.3.

If the key is some prefix of a key binding which already exists in the specified place, then the new one will override the old one, effectively deleting it.

ext:do-alpha-key-events is useful for setting up bindings in certain new modes.

command-bindings *command* [*Function*]

This function returns a list of the places where *command* is bound. A place is specified as a list of the key (always a vector), the kind of binding, and where (either the mode or buffer to which the binding is local, or **nil** if it is a global).

delete-key-binding *key* **&optional** *kind where* [*Function*]

This function removes the binding of *key* in some place. *Key* is either a key-event or a sequence of key-events. *kind* is the kind of binding to delete, one of **:global** (the default), **:mode** or **:buffer**. If *kind* is **:mode**, *where* is the mode name, and if *kind* is **:buffer**, then *where* is the buffer.

This function signals an error if *key* is unbound.

This processes *key* for key translations before deleting the binding. See section 7.2.3.

get-command *key* **&optional** *kind where* [*Function*]

This function returns the command bound to *key*, returning **nil** if it is unbound. *Key* is either a key-event or a sequence of key-events. If *key* is an initial subsequence of some keys, then this returns the keyword **:prefix**. There are four cases of *kind*:

:current Return the current binding of *key* using the current buffer's search list. If there are any transparent key bindings for *key*, then they are returned in a list as a second value.

:global Return the global binding of *key*. This is the default.

:mode Return the binding of *key* in the mode named *where*.

:buffer Return the binding of *key* local to the buffer *where*.

This processes *key* for key translations before looking for any binding. See section 7.2.3.

map-bindings *function kind* **&optional** *where* [*Function*]

This function maps over the key bindings in some place. For each binding, this passes *function* the key and the command bound to it. *Kind* and *where* are the same as in **bind-key**. The key is not guaranteed to remain valid after a given iteration.

7.2.3. Key Translation

Key translation is a process that the command interpreter applies to keys before doing anything else. There are two kinds of key translations: substitution and bit-prefix. In either case, the command interpreter translates a key when a specified key-event sequence appears in a key.

In a substitution translation, the system replaces the matched subsequence with another key-event sequence. Key translation is not recursively applied to the substituted key-events.

In a bit-prefix translation, the system removes the matched subsequence and effectively sets the specified bits in the next key-event in the key.

While translating a key, if the system encounters an incomplete final subsequence of key-events, it aborts the translation process. This happens when those last key-events form a prefix of some translation. It also happens when they translate to a bit-prefix, but there is no following key-event to which the system can apply the indicated modifier. If there is a binding for this partially untranslated key, then the command interpreter will invoke that command; otherwise, it will wait for the user to type more key-events.

key-translation *key* [*Function*]

This form is **setf**'able and allows users to register key translations that the command interpreter will use as users type key-events.

This function returns the key translation for *key*, returning **nil** if there is none. *Key* is either a key-event or a sequence of key-events. If *key* is a prefix of a translation, then this returns **:prefix**.

A key translation is either a key or modifier specification. The bits translations have a list form: **(:bits {***bit-name***}*)**.

Whenever *key* appears as a subsequence of a key argument to the binding manipulation functions, that portion will be replaced with the translation.

7.2.4. Transparent Key Bindings

Key bindings local to a mode may be *transparent*. A transparent key binding does not shadow less local key bindings, but rather indicates that the bound command should be invoked before the first normal key binding. Transparent key bindings are primarily useful for implementing minor modes such as auto fill and word abbreviation. There may be several transparent key bindings for a given key, in which case all of the commands bound are invoked in the order they were found. If there no normal key binding for a key typed, then the command interpreter acts as though the key is unbound even if there are transparent key bindings.

The **:transparent-p** argument to **defmode** (page 36) determines whether the key bindings in a mode are transparent or not.

7.2.5. Interactive

Hemlock supports keyboard macros. A user may enter a mode where the editor records his actions, and when the user exits this mode, the command Last Keyboard Macro plays back the actions. Some commands behave differently when invoked as part of the definition of a keyboard macro. For example, when used in a keyboard macro, a command that **message**'s useless user confirmation will slow down the repeated invocations of Last Keyboard Macro because the command will pause on each execution to make sure the user sees the message. This can be eliminated with the use of **interactive**. As another example, some commands conditionally signal an editor-error versus simply beeping the device depending on whether it executes on behalf of the user or a keyboard macro.

interactive [*Function*]

This returns **t** when the user invoked the command directly.

7.3. Command Types

In many editors the behavior of a command depends on the kind of command invoked before it. Hemlock provides a mechanism to support this known as *command type*.

last-command-type [*Function*]

This returns the command type of the last command invoked. If this is set with **setf**, the supplied value becomes the value of **last-command-type** until the next command completes. If the previous command did not set **last-command-type**, then its value is **nil**. Normally a command type is a keyword. The command type is not cleared after a command is invoked due to a transparent key binding.

7.4. Command Arguments

There are three ways in which a command may be invoked: It may be bound to a key which has been typed, it may be invoked as an extended command, or it may be called as a Lisp function. Ideally commands should be written in such a way that they will behave sensibly no matter which way they are invoked. The functions which implement commands must obey certain conventions about argument passing if the command is to function properly.

7.4.1. The Prefix Argument

Whenever a command is invoked it is passed as its first argument what is known as the *prefix argument*. The prefix argument is always either an integer or **`nil`**. When a command uses this value it is usually as a repeat count, or some conceptually similar function.

`prefix-argument` [*Function*]

> This function returns the current value of the prefix argument. When set with **`setf`**, the new value becomes the prefix argument for the next command.

If the prefix argument is not set by the previous command then the prefix argument for a command is **`nil`**. The prefix argument is not cleared after a command is invoked due to a transparent key binding.

7.4.2. Lisp Arguments

It is often desirable to call commands from Lisp code, in which case arguments which would otherwise be prompted for are passed as optional arguments following the prefix argument. A command should prompt for any arguments not supplied.

7.5. Recursive Edits

`use-buffer` *buffer* {*form*}* [*Macro*]

> The effect of this is similar to setting the current-buffer to *buffer* during the evaluation of *forms*. There are restrictions placed on what the code can expect about its environment. In particular, the value of any global binding of a Hemlock variable which is also a mode local variable of some mode is ill-defined; if the variable has a global binding it will be bound, but the value may not be the global value. It is also impossible to nest **`use-buffer`**'s in different buffers. The reason for using **`use-buffer`** is that it may be significantly faster than changing **`current-buffer`** to *buffer* and back.

`recursive-edit &optional` *handle-abort* [*Function*]
Enter Recursive Edit Hook [Hemlock *Variable*]

> **`recursive-edit`** invokes the command interpreter. The command interpreter will read from the keyboard and invoke commands until it is terminated with either **`exit-recursive-edit`** or **`abort-recursive-edit`**.
>
> Normally, an editor-error or **C-g** aborts the command in progress and returns control to the top-level command loop. If **`recursive-edit`** is used with *handle-abort* true, then **`editor-error`** or **C-g** will only abort back to the recursive command loop.
>
> Before the command interpreter is entered the hook Enter Recursive Edit Hook is invoked.

in-recursive-edit [*Function*]

This returns whether the calling point is dynamically within a recursive edit context.

exit-recursive-edit &optional *values-list* [*Function*]

Exit Recursive Edit Hook [Hemlock *Variable*]

exit-recursive-edit exits a recursive edit returning as multiple values each element of *values-list*, which defaults to **nil**. This invokes Exit Recursive Edit Hook after exiting the command interpreter. If no recursive edit is in progress, then this signals an error.

abort-recursive-edit &rest *args* [*Function*]

Abort Recursive Edit Hook [Hemlock *Variable*]

abort-recursive-edit terminates a recursive edit by applying **editor-error** (page 60) to *args* after exiting the command interpreter. This invokes Abort Recursive Edit Hook with *args* before aborting the recursive edit . If no recursive edit is in progress, then this signals an error.

Chapter 8

Modes

A mode is a collection of Hemlock values which may be present in the current environment (page 21) depending on the editing task at hand. Examples of typical modes are Lisp, for editing Lisp code, and Echo Area, for prompting in the echo area.

8.1. Mode Hooks

When a mode is added to or removed from a buffer, its *mode hook* is invoked. The hook functions take two arguments, the buffer involved and **`t`** if the mode is being added or **`nil`** if it is being removed.

Mode hooks are typically used to make a mode do something additional to what it usually does. One might, for example, make a text mode hook that turned on auto-fill mode when you entered.

8.2. Major and Minor Modes

There are two kinds of modes, *major* modes and *minor* modes. A buffer always has exactly one major mode, but it may have any number of minor modes. Major modes may have mode character attributes while minor modes may not.

A major mode is usually used to change the environment in some major way, such as to install special commands for editing some language. Minor modes generally change some small attribute of the environment, such as whether lines are automatically broken when they get too long. A minor mode should work regardless of what major mode and minor modes are in effect.

Default Modes (initial value **`("Fundamental" "Save")`**) [Hemlock *Variable*]

This variable contains a list of mode names which are instantiated in a buffer when no other information is available.

`*mode-names*` [*Variable*]

Holds a string-table of the names of all the modes.

Illegal [*Command*]

This is a useful command to bind in modes that wish to shadow global bindings by making them effectively illegal. Also, although less likely, minor modes may shadow major mode bindings with this. This command calls **`editor-error`**.

8.3. Mode Functions

`defmode` *name* **`&key :setup-function :cleanup-function :major-p :precedence :transparent-p :documentation`** [*Function*]

This function defines a new mode named *name*, and enters it in **`*mode-names*`** (page 35). If *major-p* is supplied and is not **`nil`** then the mode is a major mode; otherwise it is a minor mode.

Setup-function and *cleanup-function* are functions which are invoked with the buffer affected, after the mode is turned on, and before it is turned off, respectively. These functions typically are used to make buffer-local key or variable bindings and to remove them when the mode is turned off.

Precedence is only meaningful for a minor mode. The precedence of a minor mode determines the order in which it in a buffer's list of modes. When searching for values in the current environment, minor modes are searched in order, so the precedence of a minor mode determines which value is found when there are several definitions.

Transparent-p determines whether key bindings local to the defined mode are transparent. Transparent key bindings are invoked in addition to the first normal key binding found rather than shadowing less local key bindings.

Documentation is some introductory text about the mode. Commands such as Describe Mode use this.

`mode-documentation` *name* [*Function*]

This function returns the documentation for the mode named *name*.

`buffer-major-mode` *buffer* [*Function*]
Buffer Major Mode Hook [Hemlock *Variable*]

`buffer-major-mode` returns the name of *buffer*'s major mode. The major mode may be changed with **`setf`**; then Buffer Major Mode Hook is invoked with *buffer* and the new mode.

`buffer-minor-mode` *buffer name* [*Function*]
Buffer Minor Mode Hook [Hemlock *Variable*]

`buffer-minor-mode` returns **`t`** if the minor mode *name* is active in *buffer*, **`nil`** otherwise. A minor mode may be turned on or off by using **`setf`**; then Buffer Minor Mode Hook is invoked with *buffer*, *name* and the new value.

`mode-variables` *name* [*Function*]

Returns the string-table of mode local variables.

`mode-major-p` *name* [*Function*]

Returns **`t`** if *name* is the name of a major mode, or **`nil`** if it is the name of a minor mode. It is an error for *name* not to be the name of a mode.

Chapter 9

Character Attributes

9.1. Introduction

Character attributes provide a global database of information about characters. This facility is similar to, but more general than, the *syntax tables* of other editors such as **EMACS**. For example, you should use character attributes for commands that need information regarding whether a character is *whitespace* or not. Use character attributes for these reasons:

1. If this information is all in one place, then it is easy the change the behavior of the editor by changing the syntax table, much easier than it would be if character constants were wired into commands.
2. This centralization of information avoids needless duplication of effort.
3. The syntax table primitives are probably faster than anything that can be written above the primitive level.

Note that an essential part of the character attribute scheme is that *character attributes are global and are there for the user to change.* Information about characters which is internal to some set of commands (and which the user should not know about) should not be maintained as a character attribute. For such uses various character searching abilities are provided by the function **find-pattern** (page 20).

syntax-char-code-limit [*Constant*]

The exclusive upper bound on character codes which are significant in the character attribute functions. Font and bits are always ignored.

9.2. Character Attribute Names

As for Hemlock variables, character attributes have a user visible string name, but are referred to in Lisp code as a symbol. The string name, which is typically composed of capitalized words separated by spaces, is translated into a keyword by replacing all spaces with hyphens and interning this string in the keyword package. The attribute named Ada Syntax would thus become **:ada-syntax**.

character-attribute-names [*Variable*]

Whenever a character attribute is defined, its name is entered in this string table (page 69), with the corresponding keyword as the value.

9.3. Character Attribute Functions

`defattribute` *name documentation* **`&optional`** *type initial-value* [*Function*]

This function defines a new character attribute with *name*, a simple-string. Character attribute operations take attribute arguments as a keyword whose name is *name* uppercased with spaces replaced by hyphens.

Documentation describes the uses of the character attribute.

Type, which defaults to **`(mod 2)`**, specifies what type the values of the character attribute are. Values of a character attribute may be of any type which may be specified to **`make-array`**. *Initial-value* (default **`0`**) is the value which all characters will initially have for this attribute.

`character-attribute-name` *attribute* [*Function*]
`character-attribute-documentation` *attribute* [*Function*]

These functions return the name or documentation for *attribute*.

`character-attribute` *attribute character* [*Function*]
Character Attribute Hook [Hemlock *Variable*]

`character-attribute` returns the value of *attribute* for *character*. This signals an error if *attribute* is undefined.

`setf` will set a character's attributes. This **`setf`** method invokes the functions in Character Attribute Hook on the attribute and character before it makes the change.

If *character* is **`nil`**, then the value of the attribute for the beginning or end of the buffer can be accessed or set. The buffer beginning and end thus become a sort of fictitious character, which simplifies the use of character attributes in many cases.

`character-attribute-p` *symbol* [*Function*]

This function returns **`t`** if *symbol* is the name of a character attribute, **`nil`** otherwise.

`shadow-attribute` *attribute character value mode* [*Function*]
Shadow Attribute Hook [Hemlock *Variable*]

This function establishes *value* as the value of *character*'s *attribute* attribute when in the mode *mode*. *Mode* must be the name of a major mode. Shadow Attribute Hook is invoked with the same arguments when this function is called. If the value for an attribute is set while the value is shadowed, then only the shadowed value is affected, not the global one.

`unshadow-attribute` *attribute character mode* [*Function*]
Unshadow Attribute Hook [Hemlock *Variable*]

Make the value of *attribute* for *character* no longer be shadowed in *mode*. Unshadow Attribute Hook is invoked with the same arguments when this function is called.

`find-attribute` *mark attribute* **`&optional`** *test* [*Function*]
`reverse-find-attribute` *mark attribute* **`&optional`** *test* [*Function*]

These functions find the next (or previous) character with some value for the character attribute *attribute* starting at *mark*. They pass *Test* one argument, the value of *attribute* for the character tested. If the test succeeds, then these routines modify *mark* to point before (after for **`reverse-find-attribute`**) the character which satisfied the test. If no characters satisfy the test, then these return **`nil`**, and *mark* remains unmodified. *Test* defaults to **`not zerop`**. There is no guarantee that the test is applied in any particular fashion, so it should have no side effects and depend only on its argument.

9.4. Character Attribute Hooks

It is often useful to use the character attribute mechanism as an abstract interface to other information about characters which in fact is stored elsewhere. For example, some implementation of Hemlock might decide to define a Print Representation attribute which controls how a character is displayed on the screen.

To make this easy to do, each attribute has a list of hook functions which are invoked with the attribute, character and new value whenever the current value changes for any reason.

`character-attribute-hooks` *attribute* [*Function*]

Return the current hook list for *attribute*. This may be set with **`setf`**. The **`add-hook`** and **`remove-hook`** (page 25) macros should be used to manipulate these lists.

9.5. System Defined Character Attributes

These are predefined in Hemlock:

Whitespace A value of **`1`** indicates the character is whitespace.

Word Delimiter A value of **`1`** indicates the character separates words (see section 15.3).

Digit A value of **`1`** indicates the character is a base ten digit. This may be shadowed in modes or buffers to mean something else.

Space This is like Whitespace, but it should not include **Newline**. Hemlock uses this primarily for handling indentation on a line.

Sentence Terminator
A value of **`1`** indicates these characters terminate sentences (see section 15.3).

Sentence Closing Char
A value of **`1`** indicates these delimiting characters, such as **"** or **)**, may follow a Sentence Terminator (see section 15.3).

Paragraph Delimiter
A value of **`1`** indicates these characters delimit paragraphs when they begin a line (see section 15.3).

Page Delimiter A value of **`1`** indicates this character separates logical pages (see section 15.4) when it begins a line.

Scribe Syntax This uses the following symbol values:

`nil`	These characters have no interesting properties.
`:escape`	This is **@** for the Scribe formatting language.
`:open-paren`	These characters begin delimited text.
`:close-paren`	These characters end delimited text.
`:space`	These characters can terminate the name of a formatting command.
`:newline`	These characters can terminate the name of a formatting command.

Lisp Syntax This uses symbol values from the following:

`nil`	These characters have no interesting properties.
`:space`	These characters act like whitespace and should not include **Newline**.
`:newline`	This is the **Newline** character.
`:open-paren`	This is **(** character.
`:close-paren`	This is **)** character.

`:prefix` This is a character that is a part of any form it precedes — for example, the single quote, **`'`**.

`:string-quote` This is the character that quotes a string literal, **`"`**.

`:char-quote` This is the character that escapes a single character, **`\`**.

`:comment` This is the character that makes a comment with the rest of the line, **`;`**.

`:constituent` These characters are constitute symbol names.

Chapter 10

Controlling the Display

10.1. Windows

A window is a mechanism for displaying part of a buffer on some physical device. A window is a way to view a buffer but is not synonymous with one; a buffer may be viewed in any number of windows. A window may have a *modeline* which is a line of text displayed across the bottom of a window to indicate status information, typically related to the buffer displayed.

10.2. The Current Window

`current-window` [*Function*]
Set Window Hook [Hemlock *Variable*]

`current-window` returns the window in which the cursor is currently displayed. The cursor always tracks the buffer-point of the corresponding buffer. If the point is moved to a position which would be off the screen the recentering process is invoked. Recentering shifts the starting point of the window so that the point is once again displayed. The current window may be changed with **`setf`**. Before the current window is changed, the hook Set Window Hook is invoked with the new value.

`*window-list*` [*Variable*]

Holds a list of all the window objects made with **`make-window`** (page 41).

10.3. Window Functions

`make-window` *mark* **`&key :modelinep :window :ask-user`** [*Function*]
`:x :y :width :height`
Default Window Width [Hemlock *Variable*]
Default Window Height [Hemlock *Variable*]
Make Window Hook [Hemlock *Variable*]

`make-window` returns a window displaying text starting at *mark*, which must point into a buffer. If it could not make a window on the device, it returns nil. The default action is to make the new window a proportion of the **`current-window`**'s height to make room for the new window.

Modelinep specifies whether the window should display buffer modelines.

Window is a device dependent window to be used with the Hemlock window. The device may not support this argument. *Window* becomes the parent window for a new group of windows that behave in a stack orientation as windows do on the terminal.

If *ask-user* is non-**nil**, Hemlock prompts the user for the missing dimensions (*x*, *y*, *width*, and *height*) to make a new group of windows, as with the *window* argument. The device may not support this argument. Non-null values other than **t** may have device dependent meanings. *X* and *y* are in pixel units, but *width* and *height* are characters units. Default Window Width and Default Window Height are the default values for the *width* and *height* arguments.

Proportion determines what proportion of the **current-window**'s height the new window will use. The **current-window** retains whatever space left after accommodating the new one. The default is to split the window in half.

This invokes Make Window Hook with the new window.

windowp *window* [*Function*]

This function returns **t** if *window* is a **window** object, otherwise **nil**.

delete-window *window* [*Function*]
Delete Window Hook [Hemlock *Variable*]

delete-window makes *window* go away, first invoking Delete Window Hook with *window*.

window-buffer *window* [*Function*]
Window Buffer Hook [Hemlock *Variable*]

window-buffer returns the buffer from which the window displays text. This may be changed with **setf**, in which case the hook Window Buffer Hook is invoked beforehand with the window and the new buffer.

window-display-start *window* [*Function*]
window-display-end *window* [*Function*]

window-display-start returns the mark that points before the first character displayed in *window*. Note that if *window* is the current window, then moving the start may not prove much, since recentering may move it back to approximately where it was originally.

window-display-end is similar, but points after the last character displayed. Moving the end is meaningless, since redisplay always moves it to after the last character.

window-display-recentering *window* [*Function*]

This function returns whether redisplay will ensure the buffer's point of *window*'s buffer is visible after redisplay. This is **setf**'able, and changing *window*'s buffer sets this to **nil** via Window Buffer Hook.

window-point *window* [*Function*]

This function returns as a mark the position in the buffer where the cursor is displayed. This may be set with **setf**. If *window* is the current window, then setting the point will have little effect; it is forced to track the buffer point. When the window is not current, the window point is the position that the buffer point will be moved to when the window becomes current.

center-window *window mark* [*Function*]

This function attempts to adjust window's display start so the that *mark* is vertically centered within the window.

scroll-window *window n* [*Function*]

This function scrolls the window down *n* display lines; if *n* is negative scroll up. Leave the cursor at the same text position unless we scroll it off the screen, in which case the cursor is moved to the end of the window closest to its old position.

displayed-p *mark window* [*Function*]

Returns **t** if either the character before or the character after *mark* is being displayed in *window*, or **nil** otherwise.

window-height *window* [*Function*]
window-width *window* [*Function*]

Height or width of the area of the window used for displaying the buffer, in character positions. These values may be changed with **setf**, but the setting attempt may fail, in which case nothing is done.

next-window *window* [*Function*]
previous-window *window* [*Function*]

Return the next or previous window of *window*. The exact meaning of next and previous depends on the device displaying the window. It should be possible to cycle through all the windows displayed on a device using either next or previous (implying that these functions wrap around.)

10.4. Cursor Positions

A cursor position is an absolute position within a window's coordinate system. The origin is in the upper-left-hand corner and the unit is character positions.

mark-to-cursorpos *mark window* [*Function*]

Returns as multiple values the **X** and **Y** position on which *mark* is being displayed in *window*, or **nil** if it is not within the bounds displayed.

cursorpos-to-mark *X Y window* [*Function*]

Returns as a mark the text position which corresponds to the given (*X*, *Y*) position within window, or **nil** if that position does not correspond to any text within *window*.

last-key-event-cursorpos [*Function*]

Interprets mouse input. It returns as multiple values the (*X*, *Y*) position and the window where the pointing device was the last time some key event happened. If the information is unavailable, this returns **nil**.

mark-column *mark* [*Function*]

This function returns the *X* position at which *mark* would be displayed, supposing its line was displayed on an infinitely wide screen. This takes into consideration strange characters such as tabs.

move-to-column *mark column* **&optional** *line* [*Function*]

This function is analogous to **move-to-position** (page 5), except that it moves *mark* to the position on *line* which corresponds to the specified *column*. *Line* defaults to the line that *mark* is currently on. If the line would not reach to the specified column, then **nil** is returned and *mark* is not modified. Note that since a character may be displayed on more than one column on the screen, several different values of *column* may cause *mark* to be moved to the same position.

show-mark *mark window time* [*Function*]

This function highlights the position of *mark* within *window* for *time* seconds, possibly by moving the cursor there. The wait may be aborted if there is pending input. If *mark* is positioned outside the text displayed by *window*, then this returns **nil**, otherwise **t**.

10.5. Redisplay

Redisplay translates changes in the internal representation of text into changes on the screen. Ideally this process finds the minimal transformation to make the screen correspond to the text in order to maximize the speed of redisplay.

redisplay [*Function*]
Redisplay Hook [Hemlock *Variable*]

redisplay executes the redisplay process, and Hemlock typically invokes this whenever it looks for input. The redisplay process frequently checks for input, and if it detects any, it aborts. The return value is interpreted as follows:

- **nil** No update was needed.
- **t** Update was needed, and completed successfully.
- **:editor-input** Update is needed, but was aborted due to pending input.

This function invokes the functions in Redisplay Hook on the current window after computing screen transformations but before executing them. After invoking the hook, this recomputes the redisplay and then executes it on the current window.

For the current window and any window with **window-display-recentering** set, **redisplay** ensures the buffer's point for the window's buffer is visible after redisplay.

redisplay-all [*Function*]

This causes all editor windows to be completely redisplayed. For the current window and any window with **window-display-recentering** set, this ensures the buffer's point for the window's buffer is visible after redisplay. The return values are the same as for redisplay, except that **nil** is never returned.

editor-finish-output *window* [*Function*]

This makes sure the editor is synchronized with respect to redisplay output to *window*. This may do nothing on some devices.

Chapter 11

Logical Key-Events

11.1. Introduction

Some primitives such as **`prompt-for-key`** (page 50) and commands such as EMACS query replace read key-events directly from the keyboard instead of using the command interpreter. To encourage consistency between these commands and to make them portable and easy to customize, there is a mechanism for defining *logical key-events*.

A logical key-event is a keyword which stands for some set of key-events. The system globally interprets these key-events as indicators a particular action. For example, the **`:help`** logical key-event represents the set of key-events that request help in a given Hemlock implementation. This mapping is a many-to-many mapping, not one-to-one, so a given logical key-event may have multiple corresponding actual key-events. Also, any key-event may represent different logical key-events.

11.2. Logical Key-Event Functions

`*logical-key-event-names*` [*Variable*]

This variable holds a string-table mapping all logical key-event names to the keyword identifying the logical key-event.

`define-logical-key-event` *string-name documentation* [*Function*]

This function defines a new logical key-event with name *string-name*, a simple-string. Logical key-event operations take logical key-events arguments as a keyword whose name is *string-name* uppercased with spaces replaced by hyphens.

Documentation describes the action indicated by the logical key-event.

`logical-key-event-key-events` *keyword* [*Function*]

This function returns the list of key-events representing the logical key-event *keyword*.

`logical-key-event-name` *keyword* [*Function*]
`logical-key-event-documentation` *keyword* [*Function*]

These functions return the string name and documentation given to **`define-logical-key-event`** for logical key-event *keyword*.

`logical-key-event-p` *key-event keyword* [*Function*]

This function returns **`t`** if *key-event* is the logical key-event *keyword*. This is **`setf`**'able establishing or disestablishing key-events as particular logical key-events. It is a error for *keyword* to be an undefined logical key-event.

11.3. System Defined Logical Key-Events

There are many default logical key-events, some of which are used by functions documented in this manual. If a command wants to read a single key-event command that fits one of these descriptions then the key-event read should be compared to the corresponding logical key-event instead of explicitly mentioning the particular key-event in the code. In many cases you can use the **`command-case`** (page 48) macro. It makes logical key-events easy to use and takes care of prompting and displaying help messages.

`:yes` Indicates the prompter should take the action under consideration.

`:no` Indicates the prompter should NOT take the action under consideration.

`:do-all` Indicates the prompter should repeat the action under consideration as many times as possible.

`:do-once` Indicates the prompter should execute the action under consideration once and then exit.

`:exit` Indicates the prompter should terminate its activity in a normal fashion.

`:abort` Indicates the prompter should terminate its activity without performing any closing actions of convenience, for example.

`:keep` Indicates the prompter should preserve something.

`:help` Indicates the prompter should display some help information.

`:confirm` Indicates the prompter should take any input provided or use the default if the user entered nothing.

`:quote` Indicates the prompter should take the following key-event as itself without any sort of command interpretation.

`:recursive-edit`
Indicates the prompter should enter a recursive edit in the current context.

`:cancel` Indicates the prompter should cancel the effect of a previous key-event input.

`:forward-search`
Indicates the prompter should search forward in the current context.

`:backward-search`
Indicates the prompter should search backward in the current context.

Define a new logical key-event whenever:

1. The key-event concerned represents a general class of actions, and several commands may want to take a similar action of this type.
2. The exact key-event a command implementor chooses may generate violent taste disputes among users, and then the users can trivially change the command in their init files.
3. You are using **`command-case`** which prevents implementors from specifying non-standard characters for dispatching in otherwise possibly portable code, and you can define and set the logical key-event in a site dependent file where you can mention implementation dependent characters.

Chapter 12

The Echo Area

Hemlock provides a number of facilities for displaying information and prompting the user for it. Most of these work through a small window displayed at the bottom of the screen. This is called the echo area and is supported by a buffer and a window. This buffer's modeline (see section 3.3) is referred to as the status line, which, unlike other buffers' modelines, is used to show general status about the editor, Lisp, or world.

Default Status Line Fields [Hemlock *Variable*]

This is the initial list of modeline-field objects stored in the echo area buffer.

Echo Area Height (initial value **3**) [Hemlock *Variable*]

This variable determines the initial height in lines of the echo area window.

12.1. Echo Area Functions

It is considered poor taste to perform text operations on the echo area buffer to display messages; the **message** function should be used instead. A command must use this function or set **buffer-modified** (page 12) for the Echo Area buffer to **nil** to cause Hemlock to leave text in the echo area after the command's execution.

clear-echo-area [*Function*]

Clears the echo area.

message *control-string* **&rest** *format-arguments* [*Function*]
loud-message *control-string* **&rest** *format-arguments* [*Function*]
Message Pause (initial value **0.5**) [Hemlock *Variable*]

Displays a message in the echo area. The message is always displayed on a fresh line. **message** pauses for Message Pause seconds before returning to assure that messages are not displayed too briefly to be seen. Because of this, **message** is the best way to display text in the echo area.

loud-message is like **message**, but it first clears the echo area and beeps.

echo-area-window [*Variable*]
echo-area-buffer [*Variable*]

echo-area-buffer contains the buffer object for the echo area, which is named Echo Area. This buffer is usually in Echo Area mode. **echo-area-window** contains a window displaying **echo-area-buffer**. Its modeline is the status line, see the beginning of this chapter.

`*echo-area-stream*` [*Variable*]

This is a buffered Hemlock output stream (59) which inserts text written to it at the point of the echo area buffer. Since this stream is buffered a **`force-output`** must be done when output is complete to assure that it is displayed.

12.2. Prompting Functions

Most of the prompting functions accept the following keyword arguments:

`:must-exist` If **`:must-exist`** has a non-**`nil`** value then the user is prompted until a valid response is obtained. If **`:must-exist`** is **`nil`** then return as a string whatever is input. The default is **`t`**.

`:default` If null input is given when the user is prompted then this value is returned. If no default is given then some input must be given before anything interesting will happen.

`:default-string`
If a **`:default`** is given then this is a string to be printed to indicate what the default is. The default is some representation of the value for **`:default`**, for example for a buffer it is the name of the buffer.

`:prompt` This is the prompt string to display.

`:help` This is similar to **`:prompt`**, except that it is displayed when the help command is typed during input.

This may also be a function. When called with no arguments, it should either return a string which is the help text or perform some action to help the user, returning **`nil`**.

`prompt-for-buffer &key :prompt :help :must-exist :default :default-string` [*Function*]

Prompts with completion for a buffer name and returns the corresponding buffer. If *must-exist* is **`nil`**, then it returns the input string if it is not a buffer name. This refuses to accept the empty string as input when **`:default`** and **`:default-string`** are **`nil`**. **`:default-string`** may be used to supply a default buffer name when **`:default`** is **`nil`**, but when **`:must-exist`** is non-**`nil`**, it must name an already existing buffer.

`command-case` **(**{*key value*}*) {(({(*tag*}*) | *tag*} *help* {*form*}*)}* [*Macro*]

This macro is analogous to the Common Lisp **`case`** macro. Commands such as Query Replace use this to get a key-event, translate it to a character, and then to dispatch on the character to some case. In addition to character dispatching, this supports logical key-events (page 45) by using the input key-event directly without translating it to a character. Since the description of this macro is rather complex, first consider the following example:

```
(defcommand "Save All Buffers" (p)
  "Give the User a chance to save each modified buffer."
  "Give the User a chance to save each modified buffer."
  (dolist (b *buffer-list*)
    (select-buffer-command () b)
    (when (buffer-modified b)
      (command-case (:prompt "Save this buffer: [Y] "
                     :help "Save buffer, or do something else:")
        ((:yes :confirm)
         "Save this buffer and go on to the next."
         (save-file-command () b))
        (:no "Skip saving this buffer, and go on to the next.")
        (:recursive-edit
         "Go into a recursive edit in this buffer."
         (do-recursive-edit) (reprompt))
        ((:exit #\p) "Punt this silly loop."
         (return nil))))))
```

command-case prompts for a key-event and then executes the code in the first branch with a logical key-event or a character (called *tags*) matching the input. Each character must be a standard-character, one that satisfies the Common Lisp **standard-char-p** predicate, and the dispatching mechanism compares the input key-event to any character tags by mapping the key-event to a character with **ext:key-event-char**. If the tag is a logical key-event, then the search for an appropriate case compares the key-event read with the tag using **logical-key-event-p**.

All uses of **command-case** have two default cases, **:help** and **:abort**. You can override these easily by specifying your own branches that include these logical key-event tags. The **:help** branch displays in a pop-up window the a description of the valid responses using the variously specified help strings. The **:abort** branch signals an editor-error.

The *key*/*value* arguments control the prompting. The following are valid values:

:help The default **:help** case displays this string in a pop-up window. In addition it formats a description of the valid input including each case's *help* string.

:prompt This is the prompt used when reading the key-event.

:change-window
If this is non-nil (the default), then the echo area window becomes the current window while the prompting mechanism reads a key-event. Sometimes it is desirable to maintain the current window since it may be easier for users to answer the question if they can see where the current point is.

:bind This specifies a variable to which the prompting mechanism binds the input key-event. Any case may reference this variable. If you wish to know what character corresponds to the key-event, use **ext:key-event-char**.

Instead of specifying a tag or list of tags, you may use **t**. This becomes the default branch, and its forms execute if no other branch is taken, including the default **:help** and **:abort** cases. This option has no *help* string, and the default **:help** case does not describe the default branch. Every **command-case** has a default branch; if none is specified, the macro includes one that **system:beep**'s and **reprompt**'s (see below).

Within the body of **command-case**, there is a defined **reprompt** macro. It causes the prompting mechanism and dispatching mechanism to immediately repeat without further execution in the current branch.

prompt-for-key-event &key :prompt :change-window [*Function*]

This function prompts for a key-event returning immediately when the user types the next key-event. **command-case** (page 48) is more useful for most purposes. When appropriate, use logical key-events (page 45).

prompt-for-key &key :prompt :help :must-exist :default :default-string [*Function*]

This function prompts for a *key*, a vector of key-events, suitable for passing to any of the functions that manipulate key bindings (page 29). If *must-exist* is true, then the key must be bound in the current environment, and the command currently bound is returned as the second value.

prompt-for-file &key :prompt :help :must-exist :default :default-string [*Function*]

This function prompts for an acceptable filename in some system dependent fashion. "Acceptable" means that it is a legal filename, and it exists if *must-exist* is non-**nil**. **prompt-for-file** returns a Common Lisp pathname.

If the file exists as entered, then this returns it, otherwise it is merged with *default* as by **merge-pathnames**.

prompt-for-integer &key :prompt :help :must-exist :default :default-string [*Function*]

This function prompts for a possibly signed integer. If *must-exist* is **nil**, then **prompt-for-integer** returns the input as a string if it is not a valid integer.

prompt-for-keyword *string-tables* **&key :prompt :help :must-exist :default :default-string** [*Function*]

This function prompts for a keyword with completion, using the string tables in the list *string-tables*. If *must-exist* is non-**nil**, then the result must be an unambiguous prefix of a string in one of the *string-tables*, and the returns the complete string even if only a prefix of the full string was typed. In addition, this returns the value of the corresponding entry in the string table as the second value.

If *must-exist* is **nil**, then this function returns the string exactly as entered. The difference between **prompt-for-keyword** with *must-exist* **nil**, and **prompt-for-string**, is the user may complete the input using the Complete Parse and Complete Field commands.

prompt-for-expression &key :prompt :help :must-exist :default :default-string [*Function*]

This function reads a Lisp expression. If *must-exist* is **nil**, and a read error occurs, then this returns the string typed.

prompt-for-string &key :prompt :help :default :default-string [*Function*]

This function prompts for a string; this cannot fail.

prompt-for-variable &key :prompt :help :must-exist :default :default-string [*Function*]

This function prompts for a variable name. If *must-exist* is non-**nil**, then the string must be a variable *defined in the current environment,* in which case the symbol name of the variable found is returned as the second value.

prompt-for-y-or-n &key :prompt :help :must-exist :default :default-string [*Function*]

This prompts for **y**, **Y**, **n**, or **N**, returning **t** or **nil** without waiting for confirmation. When the user types a confirmation key, this returns *default* if it is supplied. If *must-exist* is **nil**, this returns whatever key-event the user first types; however, if the user types one of the above key-events, this returns **t** or **nil**. This is analogous to the Common Lisp function **y-or-n-p**.

prompt-for-yes-or-no &key :prompt :help :must-exist :default :default-string [*Function*]

This function is to **prompt-for-y-or-n** as **yes-or-no-p** is to **y-or-n-p**. "Yes" or "No" must be typed out in full and confirmation must be given.

12.3. Control of Parsing Behavior

Beep On Ambiguity (initial value **t**) [Hemlock *Variable*]

If this variable is true, then an attempt to complete a parse which is ambiguous will result in a "beep".

12.4. Defining New Prompting Functions

Prompting functions are implemented as a recursive edit in the Echo Area buffer. Completion, help, and other parsing features are implemented by commands which are bound in Echo Area Mode.

A prompting function passes information down into the recursive edit by binding a collection of special variables.

parse-verification-function [*Variable*]

The system binds this to a function that Confirm Parse (page 52) calls. It does most of the work when parsing prompted input. Confirm Parse (page 52) passes one argument, which is the string that was in ***parse-input-region*** when the user invokes the command. The function should return a list of values which are to be the result of the recursive edit, or **nil** indicating that the parse failed. In order to return zero values, a non-**nil** second value may be returned along with a **nil** first value.

parse-string-tables [*Variable*]

This is the list of **string-table**s, if any, that pertain to this parse.

parse-value-must-exist [*Variable*]

This is bound to the value of the **:must-exist** argument, and is referred to by the verification function, and possibly some of the commands.

parse-default [*Variable*]

When prompting the user, this is bound to a string representing the default object, the value supplied as the **:default** argument. Confirm Parse supplies this to the parse verification function when the ***parse-input-region*** is empty.

parse-default-string [*Variable*]

When prompting the user, if ***parse-default*** is **nil**, Hemlock displays this string as a representation of the default object; for example, when prompting for a buffer, this variable would be bound to the buffer name.

`*parse-type*` [*Variable*]

The kind of parse in progress, one of **`:file`**, **`:keyword`** or **`:string`**. This tells the completion commands how to do completion, with **`:string`** disabling completion.

`*parse-prompt*` [*Variable*]

The prompt being used for the current parse.

`*parse-help*` [*Variable*]

The help string or function being used for the current parse.

`*parse-starting-mark*` [*Variable*]

This variable holds a mark in the **`*echo-area-buffer*`** (page 47) which is the position at which the parse began.

`*parse-input-region*` [*Variable*]

This variable holds a region with **`*parse-starting-mark*`** as its start and the end of the echo-area buffer as its end. When Confirm Parse is called, the text in this region is the text that will be parsed.

12.5. Some Echo Area Commands

These are some of the Echo Area commands that coordinate with the prompting routines. Hemlock binds other commands specific to the Echo Area, but they are uninteresting to mention here, such as deleting to the beginning of the line or deleting backwards a word.

Help On Parse (bound to **Home, C-_** in Echo Area mode) [*Command*]

Display the help text for the parse currently in progress.

Complete Keyword (bound to **Escape** in Echo Area mode) [*Command*]

This attempts to complete the current region as a keyword in **`*string-tables*`**. It signals an editor-error if the input is ambiguous or incorrect.

Complete Field (bound to **Space** in Echo Area mode) [*Command*]

Similar to Complete Keyword, but only attempts to complete up to and including the first character in the keyword with a non-zero **`:parse-field-separator`** attribute. If there is no field separator then attempt to complete the entire keyword. If it is not a keyword parse then just self-insert.

Confirm Parse (bound to **Return** in Echo Area mode) [*Command*]

If **`*string-tables*`** is non-**`nil`** find the string in the region in them. Call **`*parse-verification-function*`** with the current input. If it returns a non-**`nil`** value then that is returned as the value of the parse. A parse may return a **`nil`** value if the verification function returns a non-**`nil`** second value.

Chapter 13

Files

This chapter discusses ways to read and write files at various levels — at marks, into regions, and into buffers. This also treats automatic mechanisms that affect the state of buffers in which files are read.

13.1. File Options and Type Hooks

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 -*-
```

See the *Hemlock User's Manual* for more details and predefined options.

File type hooks are executed when Hemlock reads a file into a buffer based on the type of the pathname. When the user specifies a Mode file option that turns on a major mode, Hemlock ignores type hooks. This mechanism is mostly used as a simple means for turning on some appropriate default major mode.

`define-file-option` *name* **(**_buffer value_**)** {*declaration*}* {*form*}* [*Macro*]

This defines a new file option with the string name *name*. *Buffer* and *value* specify variable names for the buffer and the option value string, and *form*'s are evaluated with these bound.

`define-file-type-hook` *type-list* **(**_buffer type_**)** {*declaration*}* {*form*}* [*Macro*]

This defines some code that **`process-file-options`** (below) executes when the file options fail to set a major mode. This associates each type, a **`simple-string`**, in *type-list* with a routine that binds *buffer* to the buffer the file is in and *type* to the type of the pathname.

`process-file-options` *buffer* **`&optional`** *pathname* [*Function*]

This checks for file options in buffer and invokes handlers if there are any. *Pathname* defaults to *buffer*'s pathname but may be **`nil`**. If there is no Mode file option that specifies a major mode, and *pathname* has a type, then this tries to invoke the appropriate file type hook. **`read-buffer-file`** calls this.

13.2. Pathnames and Buffers

There is no good way to uniquely identify buffer names and pathnames. However, Hemlock has one way of mapping pathnames to buffer names that should be used for consistency among customizations and primitives. Independent of this, Hemlock provides a means for consistently generating prompting defaults when asking the user for pathnames.

pathname-to-buffer-name *pathname* [*Function*]

This function returns a string of the form "**file-namestring directory-namestring**".

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 pathnames when needed for promting the user. Pathname Defaults is a *sticky* default. See the *Hemlock User's Manual* for more details.

buffer-default-pathname *buffer* [*Function*]

This returns Buffer Pathname if it is bound. If it is not bound, and *buffer*'s name is composed solely of alphnumeric characters, then return a pathname formed from *buffer*'s name. If *buffer*'s name has other characters in it, then return the value of Last Resort Pathname Defaults Function called on *buffer*.

13.3. File Groups

File groups provide a simple way of collecting the files that compose a system and naming that collection. Hemlock supports commands for searching, replacing, and compiling groups.

active-file-group [*Variable*]

This is the list of files that constitute the currently selected file group. If this is **nil**, then there is no current group.

do-active-group {*form*}* [*Macro*]
Group Find File (initial value **nil**) [Hemlock *Variable*]
Group Save File Confirm (initial value **t**) [Hemlock *Variable*]

do-active-group iterates over ***active-file-group*** executing the forms once for each file. While the forms are executing, the file is in the current buffer, and the point is at the beginning. If there is no active group, this signals an editor-error.

This reads each file into its own buffer using **find-file-buffer**. Since unwanted buffers may consume large amounts of memory, Group Find File controls whether to delete the buffer after executing the forms. When the variable is false, this deletes the buffer if it did not previously exist; however, regardless of this variable, if the user leaves the buffer modified, the buffer persists after the forms have completed. Whenever this processes a buffer that already existed, it saves the location of the buffer's point before and restores it afterwards.

After processing a buffer, if it is modified, **do-active-group** tries to save it. If Group Save File Confirm is non-**nil**, it asks for confirmation.

13.4. File Reading and Writing

Common Lisp pathnames are used by the file primitives. For probing, checking write dates, and so forth, all of the Common Lisp file functions are available.

read-file *pathname mark* [*Function*]

This inserts the file named by *pathname* at *mark*.

`write-file` *region pathname* **`&key :keep-backup :access :append`** [*Function*]
Keep Backup Files (initial value **`nil`**) [Hemlock *Variable*]

This function writes the contents of *region* to the file named by *pathname*. This writes *region* using a stream as if it were opened with **`:if-exists`** supplied as **`:rename-and-delete`**.

When *keep-backup*, which defaults to the value of Keep Backup Files, is non-**`nil`**, this opens the stream as if **`:if-exists`** were **`:rename`**. If *append* is non-**`nil`**, this writes the file as if it were opened with **`:if-exists`** supplied as **`:append`**.

This signals an error if both *append* and *keep-backup* are supplied as non-**`nil`**.

Access is an implementation dependent value that is suitable for setting *pathname*'s access or protection bits.

`write-buffer-file` *buffer pathname* [*Function*]
Write File Hook [Hemlock *Variable*]
Add Newline at EOF on Writing File (initial value **`:ask-user`**) [Hemlock *Variable*]

`write-buffer-file` writes *buffer* to the file named by *pathname* including the following:

- It assumes pathname is somehow related to *buffer*'s pathname: if the *buffer*'s write date is not the same as *pathname*'s, then this prompts the user for confirmation before overwriting the file.
- It consults Add Newline at EOF on Writing File (see *Hemlock User's Manual* for possible values) and interacts with the user if necessary.
- It sets Pathname Defaults, and after using **`write-file`**, marks *buffer* unmodified.
- It updates *Buffer*'s pathname and write date.
- It renames the buffer according to the new pathname if possible.
- It invokes Write File Hook.

Write File Hook is a list of functions that take the newly written buffer as an argument.

`read-buffer-file` *pathname buffer* [*Function*]
Read File Hook [Hemlock *Variable*]

`read-buffer-file` deletes *buffer*'s region and uses **`read-file`** to read *pathname* into it, including the following:

- It sets *buffer*'s write date to the file's write date if the file exists; otherwise, it **`message`**'s that this is a new file and sets *buffer*'s write date to **`nil`**.
- It moves *buffer*'s point to the beginning.
- It sets *buffer*'s unmodified status.
- It sets *buffer*'s pathname to the result of probing *pathname* if the file exists; otherwise, this function sets *buffer*'s pathname to the result of merging *pathname* with **`default-directory`**.
- It sets Pathname Defaults to the result of the previous item.
- It processes the file options.
- It invokes Read File Hook.

Read File Hook is a list functions that take two arguments — the buffer read into and whether the file existed, **`t`** if so.

find-file-buffer *pathname* [*Function*]

This returns a buffer assoicated with the *pathname*, reading the file into a new buffer if necessary. This returns a second value indicating whether a new buffer was created, **t** if so. If the file has already been read, this checks to see if the file has been modified on disk since it was read, giving the user various recovery options. This is the basis of the Find File command.

Chapter 14

Hemlock's Lisp Environment

This chapter is sort of a catch all for any functions and variables which concern Hemlock's interaction with the outside world.

14.1. Entering and Leaving the Editor

`ed &optional` *x* [*Function*]
Entry Hook [Hemlock *Variable*]

`ed` enters the editor. It is basically as specified in Common Lisp. If *x* is supplied and is a symbol, the definition of *x* is put into a buffer, and that buffer is selected. If *x* is a pathname, the file specified by *x* is visited in a new buffer. If *x* is not supplied or **`nil`**, the editor is entered in the same state as when last exited.

The Entry Hook is invoked each time the editor is entered.

`exit-hemlock &optional` *value* [*Function*]
Exit Hook [Hemlock *Variable*]

`exit-hemlock` leaves Hemlock and return to Lisp; *value* is the value to return, which defaults to **`t`**. The hook Exit Hook (page 57) is invoked before this is done.

`pause-hemlock` [*Function*]

`pause-hemlock` suspends the editor process and returns control to the shell. When the process is resumed, it will still be running Hemlock.

14.2. Keyboard Input

Keyboard input interacts with a number of other parts of the editor. Since the command loop works by reading from the keyboard, keyboard input is the initial cause of everything that happens. Also, Hemlock redisplays in the low-level input loop when there is no available input from the user.

`*editor-input*` [*Variable*]
`*real-editor-input*` [*Variable*]
Input Hook [Hemlock *Variable*]
Abort Hook [Hemlock *Variable*]

`*editor-input*` is an object on which Hemlock's I/O routines operate. You can get input, clear input, return input, and listen for input. Input appears as key-events.

`*real-editor-input*` holds the initial value of **`*editor-input*`**. This is useful for reading from the user when **`*editor-input*`** is rebound (such as within a keyboard macro.)

Hemlock invokes the functions in Input Hook each time someone reads a key-event from **`*real-editor-input*`**. These take no arguments.

`get-key-event` *editor-input* **`&optional`** *ignore-abort-attempts-p* [*Function*]

This function returns a key-event as soon as it is available on *editor-input*. *Editor-input* is either **`*editor-input*`** or **`*real-editor-input*`**. *Ignore-abort-attempts-p* indicates whether **C-g** and **C-G** throw to the editor's top-level command loop; when this is non-nil, this function returns those key-events when the user types them. Otherwise, it aborts the editor's current state, returning to the command loop.

When the user aborts, Hemlock invokes the functions in Abort Hook. These functions take no arguments. When aborting, Hemlock ignores the Input Hook.

`unget-key-event` *key-event* *editor-input* [*Function*]

This function returns *key-event* to *editor-input*, so the next invocation of **`get-key-event`** will return *key-event*. If *key-event* is **`#k"C-g"`** or **`#k"C-G"`**, then whether **`get-key-event`** returns it depends on that function's second argument. *Editor-input* is either **`*editor-input*`** or **`*real-editor-input*`**.

`clear-editor-input` *editor-input* [*Function*]

This function flushes any pending input on *editor-input*. *Editor-input* is either **`*editor-input*`** or **`*real-editor-input*`**.

`listen-editor-input` *editor-input* [*Function*]

This function returns whether there is any input available on *editor-input*. *Editor-input* is either **`*editor-input*`** or **`*real-editor-input*`**.

`editor-sleep` *time* [*Function*]

Return either after *time* seconds have elapsed or when input is available on **`*editor-input*`**.

`*key-event-history*` [*Variable*]

This is a Hemlock ring buffer (see page 70) that holds the last 60 key-events read from the keyboard.

`*last-key-event-typed*` [*Variable*]

Commands use this variable to realize the last key-event the user typed to invoke the commands. Before Hemlock ever reads any input, the value is **`nil`**. This variable usually holds the last key-event read from the keyboard, but it is also maintained within keyboard macros allowing commands to behave the same on each repetition as they did in the recording invocation.

`*input-transcript*` [*Variable*]

If this is non-**`nil`** then it should be an adjustable vector with a fill-pointer. When it is non-**`nil`**, Hemlock pushes all input read onto this vector.

14.3. Hemlock Streams

It is possible to create streams which output to or get input from a buffer. This mechanism is quite powerful and permits easy interfacing of Hemlock to Lisp.

make-hemlock-output-stream *mark* **&optional** *buffered* [*Function*]
hemlock-output-stream-p *object* [*Function*]

make-hemlock-output-stream returns a stream that inserts at the permanent mark *mark* all output directed to it. *Buffered* controls whether the stream is buffered or not, and its valid values are the following keywords:

:none	No buffering is done. This is the default.
:line	The buffer is flushed whenever a newline is written or when it is explicitly done with **force-output**.
:full	The screen is only brought up to date when it is explicitly done with **force-output**

hemlock-output-stream-p returns **t** if *object* is a **hemlock-output-stream** object.

make-hemlock-region-stream *region* [*Function*]
hemlock-region-stream-p *object* [*Function*]

make-hemlock-region-stream returns a stream from which the text in *region* can be read. **hemlock-region-stream-p** returns **t** if *object* is a **hemlock-region-stream** object.

with-input-from-region (*var region***)** {*declaration*}* {*form*}* [*Macro*]

While evaluating *form*s, binds *var* to a stream which returns input from *region*.

with-output-to-mark (*var mark* [*buffered*]**)** {*declaration*}* {*form*}* [*Macro*]

During the evaluation of the *form*s, binds *var* to a stream which inserts output at the permanent *mark*. *Buffered* has the same meaning as for **make-hemlock-output-stream**.

with-pop-up-display (*var* **&key** *height name***)** {*declaration*}* {*form*}* [*Macro*]
random-typeout-buffers [*Variable*]

This macro executes *forms* in a context with *var* bound to a stream. Hemlock collects output to this stream and tries to pop up a display of the appropriate height containing all the output. When *height* is supplied, Hemlock creates the pop-up display immediately, forcing output on line breaks. The system saves the output in a buffer named *name*, which defaults to Random Typeout. When the window is the incorrect height, the display mechanism will scroll the window with more-style prompting. This is useful for displaying information of temporary interest.

When a buffer with name *name* already exists and was not previously created by **with-pop-up-display**, Hemlock signals an error.

random-typeout-buffers is an association list mapping random typeout buffers to the streams that operate on the buffers.

14.4. Interface to the Error System

The error system interface is minimal. There is a simple editor-error condition which is a subtype of error and a convenient means for signaling them. Hemlock also provides a standard handler for error conditions while in the editor.

`editor-error-format-string` *condition* [*Function*]
`editor-error-format-arguments` *condition* [*Function*]

Handlers for editor-error conditions can access the condition object with these.

`editor-error &rest` *args* [*Function*]

This function is called to signal minor errors within Hemlock; these are errors that a normal user could encounter in the course of editing such as a search failing or an attempt to delete past the end of the buffer. This function **`signal`**'s an editor-error condition formed from *args*, which are **`nil`** or a **`format`** string possibly followed by **`format`** arguments. Hemlock invokes commands in a dynamic context with an editor-error condition handler bound. This default handler beeps or flashes (or both) the display. If the condition passed to the handler has a non-**`nil`** string slot, the handler also invokes **`message`** on it. The command in progress is always aborted, and this function never returns.

`handle-lisp-errors` {*form*}* [*Macro*]

Within the body of this macro any Lisp errors that occur are handled in some fashion more gracefully than simply dumping the user in the debugger. This macro should be wrapped around code which may get an error due to some action of the user — for example, evaluating code fragments on the behalf of and supplied by the user. Using this in a command allows the established handler to shadow the default editor-error handler, so commands should take care to signal user errors (calls to **`editor-errors`**) outside of this context.

14.5. Definition Editing

Hemlock provides commands for finding the definition of a function, macro, or command and placing the user at the definition in a buffer. This, of course, is implementation dependent, and if an implementation does not associate a source file with a routine, or if Hemlock cannot get at the information, then these commands do not work. If the Lisp system does not store an absolute pathname, independent of the machine on which the maintainer built the system, then users need a way of translating a source pathname to one that will be able to locate the source.

`add-definition-dir-translation` *dir1 dir2* [*Function*]

This maps directory pathname *dir1* to *dir2*. Successive invocations using the same *dir1* push into a translation list. When Hemlock seeks a definition source file, and it has a translation, then it tries the translations in order. This is useful if your sources are on various machines, some of which may be down. When Hemlock tries to find a translation, it first looks for translations of longer directory pathnames, finding more specific translations before shorter, more general ones.

`delete-definition-dir-translation` *dir* [*Function*]

This deletes the mapping of *dir* to all directories to which it has been mapped.

14.6. Event Scheduling

The mechanism described in this chapter is only operative when the Lisp process is actually running inside of Hemlock, within the **`ed`** function. The designers intended its use to be associated with the editor, such as with auto-saving files, reminding the user, etc.

schedule-event *time function* **&optional** *repeat* [*Function*]

This causes Hemlock to call *function* after *time* seconds have passed, optionally repeating every *time* seconds. *Repeat* defaults to **t**. This is a rough mechanism since commands can take an arbitrary amount of time to run; Hemlock invokes *function* at the first possible moment after *time* has elapsed. *Function* takes the time in seconds that has elapsed since the last time it was called (or since it was scheduled for the first invocation).

remove-scheduled-event *function* [*Function*]

This removes *function* from the scheduling queue. *Function* does not have to be in the queue.

14.7. Miscellaneous

in-lisp {*form*}* [*Function*]

This evaluates *form*'s inside **handle-lisp-errors**. It also binds ***package*** to the package named by Current Package if it is non-**nil**. Use this when evaluating Lisp code on behalf of the user.

do-alpha-chars (*var kind* **[***result*) {*form*}*] [*Macro*]

This iterates over alphabetic characters in Common Lisp binding *var* to each character in order as specified under character relations in *Common Lisp the Language*. *Kind* is one of **:lower**, **:upper**, or **:both**. When the user supplies **:both**, lowercase characters are processed first.

Chapter 15

High-Level Text Primitives

This chapter discusses primitives that operate on higher level text forms than characters and words. For English text, there are functions that know about sentence and paragraph structures, and for Lisp sources, there are functions that understand this language. This chapter also describes mechanisms for organizing file sections into *logical pages* and for formatting text forms.

15.1. Indenting Text

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 that 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. A function for Lisp mode probably moves the mark to the beginning of the line, deletes horizontal whitespace, and computes some appropriate indentation for Lisp code.

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.*

`indent-region` *region* [*Function*]
`indent-region-for-commands` *region* [*Function*]

`indent-region` invokes the value of Indent Function on every line of region. **`indent-region-for-commands`** uses **`indent-region`** but first saves the region for the Undo command.

`delete-horizontal-space` *mark* [*Function*]

This deletes all characters with a Space attribute (see section 9.5) of **`1`**.

15.2. Lisp Text Buffers

Hemlock bases its Lisp primitives on parsing a block of the buffer and annotating lines as to what kind of Lisp syntax occurs on the line or what kind of form a mark might be in (for example, string, comment, list, etc.). These do not work well if the block of parsed forms is exceeded when moving marks around these forms, but the block that gets parsed is somewhat programmable.

There is also a notion of a *top level form* which this documentation often uses synonymously with *defun*, meaning a Lisp form occurring in a source file delimited by parentheses with the opening parenthesis at the beginning of some line. The names of the functions include this inconsistency.

`pre-command-parse-check` *mark for-sure* [*Function*]
Parse Start Function (initial value **`start-of-parse-block`**) [Hemlock *Variable*]
Parse End Function (initial value **`end-of-parse-block`**) [Hemlock *Variable*]
Minimum Lines Parsed (initial value **`50`**) [Hemlock *Variable*]
Maximum Lines Parsed (initial value **`500`**) [Hemlock *Variable*]
Defun Parse Goal (initial value **`2`**) [Hemlock *Variable*]

> **`pre-command-parse-check`** calls Parse Start Function and Parse End Function on *mark* to get two marks. It then parses all the lines between the marks including the complete lines they point into. When *for-sure* is non-**`nil`**, this parses the area regardless of any cached information about the lines. Every command that uses the following routines calls this before doing so.
>
> The default values of the start and end variables use Minimum Lines Parsed, Maximum Lines Parsed, and Defun Parse Goal to determine how big a region to parse. These two functions always include at least the minimum number of lines before and after the mark passed to them. They try to include Defun Parse Goal number of top level forms before and after the mark passed them, but these functions never return marks that include more than the maximum number of lines before or after the mark passed to them.

`form-offset` *mark count* [*Function*]

> This tries to move *mark count* forms forward if positive or -*count* forms backwards if negative. *Mark* is always moved. If there were enough forms in the appropriate direction, this returns *mark*, otherwise nil.

`top-level-offset` *mark count* [*Function*]

> This tries to move *mark count* top level forms forward if positive or -*count* top level forms backwards if negative. If there were enough top level forms in the appropriate direction, this returns *mark*, otherwise nil. *Mark* is moved only if this is successful.

`mark-top-level-form` *mark1 mark2* [*Function*]

> This moves *mark1* and *mark2* to the beginning and end, respectively, of the current or next top level form. *Mark1* is used as a reference to start looking. The marks may be altered even if unsuccessful. If successful, return *mark2*, else nil. *Mark2* is left at the beginning of the line following the top level form if possible, but if the last line has text after the closing parenthesis, this leaves the mark immediately after the form.

`defun-region` *mark* [*Function*]

> This returns a region around the current or next defun with respect to *mark*. *Mark* is not used to form the region. If there is no appropriate top level form, this signals an editor-error. This calls **`pre-command-parse-check`** first.

`inside-defun-p` *mark* [*Function*]
`start-defun-p` *mark* [*Function*]

> These return, respectively, whether *mark* is inside a top level form or at the beginning of a line immediately before a character whose Lisp Syntax (see section 9.5) value is **`:opening-paren`**.

forward-up-list *mark* [*Function*]
backward-up-list *mark* [*Function*]

Respectively, these move *mark* immediately past a character whose Lisp Syntax (see section 9.5) value is **:closing-paren** or immediately before a character whose Lisp Syntax value is **:opening-paren**.

valid-spot *mark forwardp* [*Function*]

This returns **t** or **nil** depending on whether the character indicated by *mark* is a valid spot. When *forwardp* is set, use the character after mark and vice versa. Valid spots exclude commented text, inside strings, and character quoting.

defindent *name count* [*Function*]

This defines the function with *name* to have *count* special arguments. **indent-for-lisp**, the value of Indent Function (see section 15.1) in Lisp mode, uses this to specially indent these arguments. For example, **do** has two, **with-open-file** has one, etc. There are many of these defined by the system including definitions for special Hemlock forms. *Name* is a simple-string, case insensitive and purely textual (that is, not read by the Lisp reader); therefore, **"with-a-mumble"** is distinct from **"mumble:with-a-mumble"**.

15.3. English Text Buffers

This section describes some routines that understand basic English language forms.

word-offset *mark count* [*Function*]

This moves *mark count* words forward (if positive) or backwards (if negative). If *mark* is in the middle of a word, that counts as one. If there were *count* (-*count* if negative) words in the appropriate direction, this returns *mark*, otherwise nil. This always moves *mark*. A word lies between two characters whose Word Delimiter attribute value is **1** (see section 9.5).

sentence-offset *mark count* [*Function*]

This moves *mark count* sentences forward (if positive) or backwards (if negative). If *mark* is in the middle of a sentence, that counts as one. If there were *count* (-*count* if negative) sentences in the appropriate direction, this returns *mark*, otherwise nil. This always moves *mark*.

A sentence ends with a character whose Sentence Terminator attribute is **1** followed by two spaces, a newline, or the end of the buffer. The terminating character is optionally followed by any number of characters whose Sentence Closing Char attribute is **1**. A sentence begins after a previous sentence ends, at the beginning of a paragraph, or at the beginning of the buffer.

paragraph-offset *mark count* **&optional** *prefix* [*Function*]
Paragraph Delimiter Function [Hemlock *Variable*]

This moves *mark count* paragraphs forward (if positive) or backwards (if negative). If *mark* is in the middle of a paragraph, that counts as one. If there were *count* (-*count* if negative) paragraphs in the appropriate direction, this returns *mark*, otherwise nil. This only moves *mark* if there were enough paragraphs.

Paragraph Delimiter Function holds a function that takes a mark, typically at the beginning of a line, and returns whether or not the current line should break the paragraph. **default-para-delim-function** returns **t** if the next character, the first on the line, has a Paragraph Delimiter attribute value of **1**. This is typically a space, for an indented paragraph, or a newline, for a block style. Some modes require a more complicated determinant; for example, Scribe modes adds some characters to the set and special cases certain formatting commands.

Prefix defaults to Fill Prefix (see section 15.5), and the right prefix is necessary to correctly skip paragraphs. If *prefix* is non-**nil**, and a line begins with *prefix*, then the scanning process skips the prefix before invoking the Paragraph Delimiter Function. Note, when scanning for paragraph bounds, and *prefix* is non-**nil**, lines are potentially part of the paragraph regardless of whether they contain the prefix; only the result of invoking the delimiter function matters.

The programmer should be aware of an idiom for finding the end of the current paragraph. Assume **paragraphp** is the result of moving **mark** one paragraph, then the following correctly determines whether there actually is a current paragraph:

```
(or paragraphp
    (and (last-line-p mark)
         (end-line-p mark)
         (not (blank-line-p (mark-line mark)))))
```

In this example **mark** is at the end of the last paragraph in the buffer, and there is no last newline character in the buffer. **paragraph-offset** would have returned **nil** since it could not skip any paragraphs since **mark** was at the end of the current and last paragraph. However, you still have found a current paragraph on which to operate. **mark-paragraph** understands this problem.

mark-paragraph mark1 mark2 [*Function*]

This marks the next or current paragraph, setting *mark1* to the beginning and *mark2* to the end. This uses Fill Prefix (see section 15.5). *Mark1* is always on the first line of the paragraph, regardless of whether the previous line is blank. *Mark2* is typically at the beginning of the line after the line the paragraph ends on, this returns *mark2* on success. If this cannot find a paragraph, then the marks are left unmoved, and **nil** is returned.

15.4. Logical Pages

Logical pages are a way of dividing a file into coarse divisions. This is analogous to dividing a paper into sections, and Hemlock provides primitives for moving between the pages of a file and listing a directory of the page titles. Pages are separated by Page Delimiter characters (see section 9.5) that appear at the beginning of a line.

goto-page *mark n* [*Function*]

This moves *mark* to the absolute page numbered *n*. If there are less than *n* pages, it signals an editor-error. If it returns, it returns *mark*. Hemlock numbers pages starting with one for the page delimited by the beginning of the buffer and the first Page Delimiter (or the end of the buffer).

page-offset *mark n* [*Function*]

This moves mark forward *n* (-*n* backwards, if *n* is negative) Page Delimiter characters that are in the zero'th line position. If a Page Delimiter is the immediately next character after mark (or before mark, if *n* is negative), then skip it before starting. This always moves *mark*, and if there were enough pages to move over, it returns *mark*; otherwise, it returns **nil**.

page-directory *buffer* [*Function*]

This returns a list of each first non-blank line in *buffer* that follows a Page Delimiter character that is in the zero'th line position. This includes the first line of the *buffer* as the first page title. If a page is empty, then its title is the empty string.

display-page-directory *stream directory* [*Function*]

This writes the list of strings, *directory*, to *stream*, enumerating them in a field three wide. The number and string are separated by two spaces, and the first line contains headings for the page numbers and title strings.

15.5. Filling

Filling is an operation on text that breaks long lines at word boundaries before a given column and merges shorter lines together in an attempt to make each line roughly the specified length. This is different from justification which tries to add whitespace in awkward places to make each line exactly the same length. Hemlock's filling optionally inserts a specified string at the beginning of each line. Also, it eliminates extra whitespace between lines and words, but it knows two spaces follow sentences (see section 15.3).

Fill Column (initial value **75**) [Hemlock *Variable*]
Fill Prefix (initial value **nil**) [Hemlock *Variable*]

These variables hold the default values of the prefix and column arguments to Hemlock's filling primitives. If Fill Prefix is **nil**, then there is no fill prefix.

fill-region *region* **&optional** *prefix column* [*Function*]

This deletes any blank lines in region and fills it according to prefix and column. *Prefix* and *column* default to Fill Prefix and Fill Column.

fill-region-by-paragraphs *region* **&optional** *prefix column* [*Function*]

This finds paragraphs (see section 15.3) within region and fills them with **fill-region**. This ignores blank lines between paragraphs. *Prefix* and *column* default to Fill Prefix and Fill Column.

Chapter 16

Utilities

This chapter describes a number of utilities for manipulating some types of objects Hemlock uses to record information. String-tables are used to store names of variables, commands, modes, and buffers. Ring lists can be used to provide a kill ring, recent command history, or other user-visible features.

16.1. String-table Functions

String tables are similar to Common Lisp hash tables in that they associate a value with an object. There are a few useful differences: in a string table the key is always a case insensitive string, and primitives are provided to facilitate keyword completion and recognition. Any type of string may be added to a string table, but the string table functions always return **`simple-string`**'s.

A string entry in one of these tables may be thought of as being separated into fields or keywords. The interface provides keyword completion and recognition which is primarily used to implement some Echo Area commands. These routines perform a prefix match on a field-by-field basis allowing the ambiguous specification of earlier fields while going on to enter later fields. While string tables may use any **`string-char`** as a separator, the use of characters other than **space** may make the Echo Area commands fail or work unexpectedly.

`make-string-table &key :separator :initial-contents` [*Function*]

This function creates an empty string table that uses *separator* as the character, which must be a **`string-char`**, that distinguishes fields. *Initial-contents* specifies an initial set of strings and their values in the form of a dotted **`a-list`**, for example:

```
'(("Global" . t) ("Mode" . t) ("Buffer" . t))
```

`string-table-p` *string-table* [*Function*]

This function returns **`t`** if *string-table* is a **`string-table`** object, otherwise **`nil`**.

`string-table-separator` *string-table* [*Function*]

This function returns the separator character given to **`make-string-table`**.

`delete-string` *string table* [*Function*]
`clrstring` *table* [*Function*]

`delete-string` removes any entry for *string* from the **`string-table`** *table*, returning **`t`** if there was an entry. **`clrstring`** removes all entries from *table*.

`getstring` *string table* [*Function*]

This function returns as multiple values, first the value corresponding to the string if it is found and **`nil`** if it isn't, and second **`t`** if it is found and **`nil`** if it isn't.

This may be set with **`setf`** to add a new entry or to store a new value for a string. It is an error to try to insert a string with more than one field separator character occurring contiguously.

`complete-string` *string tables* [*Function*]

This function completes *string* as far as possible over the list of *tables*, returning five values. It is an error for *tables* to have different separator characters. The five return values are as follows:

- The maximal completion of the string or **`nil`** if there is none.
- An indication of the usefulness of the returned string:

  `:none` There is no completion of *string*.

  `:complete` The completion is a valid entry, but other valid completions exist too. This occurs when the supplied string is an entry as well as initial substring of another entry.

  `:unique` The completion is a valid entry and unique.

  `:ambiguous` The completion is invalid; **`get-string`** would return **`nil`** and **`nil`** if given the returned string.

- The value of the string when the completion is **`:unique`** or **`:complete`**, otherwise **`nil`**.
- An index, or nil, into the completion returned, indicating where the addition of a single field to *string* ends. The command Complete Field uses this when the completion contains the addition to *string* of more than one field.
- An index to the separator following the first ambiguous field when the completion is **`:ambiguous`** or **`:complete`**, otherwise **`nil`**.

`find-ambiguous` *string table* [*Function*]
`find-containing` *string table* [*Function*]

`find-ambiguous` returns a list in alphabetical order of all the strings in *table* matching *string*. This considers an entry as matching if each field in *string*, taken in order, is an initial substring of the entry's fields; entry may have fields remaining.

`find-containing` is similar, but it ignores the order of the fields in *string*, returning all strings in *table* matching any permutation of the fields in *string*.

`do-strings` **(***string-var value-var table* [*result*]**)** {*declaration*}* {*tag* | *statement*}* [*Macro*]

This macro iterates over the strings in *table* in alphabetical order. On each iteration, it binds *string-var* to an entry's string and *value-var* to an entry's value.

16.2. Ring Functions

There are various purposes in an editor for which a ring of values can be used, so Hemlock provides a general ring buffer type. It is used for maintaining a ring of killed regions (see section 4.3), a ring of marks (see section 3.1), or a ring of command strings which various modes and commands maintain as a history mechanism.

make-ring *length* **&optional** *delete-function* [*Function*]

Makes an empty ring object capable of holding up to *length* Lisp objects. *Delete-function* is a function that each object is passed to before it falls off the end. *Length* must be greater than zero.

ringp *ring* [*Function*]

Returns **t** if *ring* is a **ring** object, otherwise **nil**.

ring-length *ring* [*Function*]

Returns as multiple-values the number of elements which *ring* currently holds and the maximum number of elements which it may hold.

ring-ref *ring index* [*Function*]

Returns the *index*'th item in the *ring*, where zero is the index of the most recently pushed. This may be set with **setf**.

ring-push *object ring* [*Function*]

Pushes *object* into *ring*, possibly causing the oldest item to go away.

ring-pop *ring* [*Function*]

Removes the most recently pushed object from *ring* and returns it. If the ring contains no elements then an error is signalled.

rotate-ring *ring offset* [*Function*]

With a positive *offset*, rotates *ring* forward that many times. In a forward rotation the index of each element is reduced by one, except the one which initially had a zero index, which is made the last element. A negative offset rotates the ring the other way.

16.3. Undoing commands

save-for-undo *name method* **&optional** *cleanup method-undo buffer* [*Function*]

This saves information to undo a command. *Name* is a string to display when prompting the user for confirmation when he invokes the Undo command (for example, **"kill"** or **"Fill Paragraph"**). *Method* is the function to invoke to undo the effect of the command. *Method-undo* is a function that undoes the undo function, or effectively re-establishes the state immediately after invoking the command. If there is any existing undo information, this invokes the *cleanup* function; typically *method* closes over or uses permanent marks into a buffer, and the *cleanup* function should delete such references. *Buffer* defaults to the **current-buffer**, and the Undo command only invokes undo methods when they were saved for the buffer that is current when the user invokes Undo.

make-region-undo *kind name region* **&optional** *mark-or-region* [*Function*]

This handles three common cases that commands fall into when setting up undo methods, including cleanup and method-undo functions (see **save-for-undo**). These cases are indicated by the *kind* argument:

:twiddle Use this kind when a command modifies a region, and the undo information indicates how to swap between two regions — the one before any modification occurs and the resulting region. *Region* is the resulting region, and it has permanent marks into the buffer. *Mark-or-region* is a region without marks into the buffer (for example, the result of **copy-region**). As a result of calling this, a first invocation of Undo deletes *region*, saving it, and inserts *mark-or-region* where *region* used to be.

The undo method sets up for a second invocation of Undo that will undo the effect of the undo; that is, after two calls, the buffer is exactly as it was after invoking the command. This activity is repeatable any number of times. This establishes a cleanup method that deletes the two permanent marks into the buffer used to locate the modified region.

:insert Use this kind when a command has deleted a region, and the undo information indicates how to re-insert the region. *Region* is the deleted and saved region, and it does not contain marks into any buffer. *Mark-or-region* is a permanent mark into the buffer where the undo method should insert *region*. As a result of calling this, a first invocation of Undo inserts *region* at *mark-or-region* and forms a region around the inserted text with permanent marks into the buffer. This allows a second invocation of Undo to undo the effect of the undo; that is, after two calls, the buffer is exactly as it was after invoking the command. This activity is repeatable any number of times. This establishes a cleanup method that deletes either the permanent mark into the buffer or the two permanent marks of the region, depending on how many times the user used Undo.

:delete Use this kind when a command has inserted a block of text, and the undo information indicates how to delete the region. *Region* has permanent marks into the buffer and surrounds the inserted text. Leave *Mark-or-region* unspecified. As a result of calling this, a first invocation of Undo deletes *region*, saving it, and establishes a permanent mark into the buffer to remember where the *region* was. This allows a second invocation of Undo to undo the effect of the undo; that is, after two calls, the buffer is exactly as it was after invoking the command. This activity is repeatable any number of times. This establishes a cleanup method that deletes either the permanent mark into the buffer or the two permanent marks of the region, depending on how many times the user used Undo.

Name in all cases is an appropriate string indicating what the command did. This is used by Undo when prompting the user for confirmation before calling the undo method. The string used by Undo alternates between this argument and something to indicate that the user is undoing an undo.

Chapter 17

Miscellaneous

This chapter is somewhat of a catch-all for comments and features that don't fit well anywhere else.

17.1. Generic Pointer Up

Generic Pointer Up is a Hemlock command bound to mouse up-clicks. It invokes a function supplied with the interface described in this section. This command allows different commands to be bound to the same down-click in various modes with one command bound to the corresponding up-click.

`supply-generic-pointer-up-function` *function* [*Function*]

This function supplies a function that Generic Pointer Up invokes the next time it executes.

17.2. Using View Mode

View mode supports scrolling through files automatically terminating the buffer at end-of-file as well as commands for quitting the mode and popping back to the buffer that spawned the View mode buffer. Modes such as Dired and Lisp-Lib use this to view files and description of library entries.

Modes that want similar commands should use **`view-file-command`** to view a file and get a handle on the view buffer. To allow the View Return and View Quit commands to return to the originating buffer, you must set the variable View Return Function in the viewing buffer to a function that knows how to do this. Furthermore, since you now have a reference to the originating buffer, you must add a buffer local delete hook to it that will clear the view return function's reference. This needs to happen for two reasons in case the user deletes the originating buffer:

1. You don't want the return function to go to a non-existing, invalid buffer.
2. Since the viewing buffer still exists, its View Return Function buffer local variable still exists. This means the function still references the deleted originating buffer, and garbage collection cannot reclaim the memory locked down by the deleted buffer.

The following is a piece of code that could implement part of Dired View File that uses two closures to accomplish that described above:

```
(let* ((dired-buf (current-buffer))
       (buffer (view-file-command nil pathname)))
  (push #'(lambda (buffer)
            (declare (ignore buffer))
            (setf dired-buf nil))
        (buffer-delete-hook dired-buf))
  (setf (variable-value 'view-return-function :buffer buffer)
        #'(lambda ()
            (if dired-buf
                (change-to-buffer dired-buf)
                (dired-from-buffer-pathname-command nil)))))
```

The Dired buffer's delete hook clears the return function's reference to the Dired buffer. The return function tests the variable to see if it still holds a buffer when the function executes.

Chapter 18

Auxiliary Systems

This chapter describes utilities that some implementations of Hemlock may leave unprovided or unsupported.

18.1. Key-events

These routines are defined in the **"EXTENSIONS"** package since other projects have often used Hemlock's input translations for interfacing to CLX.

18.1.1. Introduction

The canonical representation of editor input is a key-event structure. Users can bind commands to keys (see section 7.2.2), which are non-zero length sequences of key-events. A key-event consists of an identifying token known as a *keysym* and a field of bits representing modifiers. Users define keysyms, integers between 0 and 65535 inclusively, by supplying names that reflect the legends on their keyboard's keys. Users define modifier names similarly, but the system chooses the bit and mask for recognizing the modifier. You can use keysym and modifier names to textually specify key-events and Hemlock keys in a **#k** syntax. The following are some examples:

```
#k"C-u"
#k"Control-u"
#k"c-m-z"
#k"control-x meta-d"
#k"a"
#k"A"
#k"Linefeed"
```

This is convenient for use within code and in init files containing **bind-key** calls.

The **#k** syntax is delimited by double quotes, but the system parses the contents rather than reading it as a Common Lisp string. Within the double quotes, spaces separate multiple key-events. A single key-event optionally starts with modifier names terminated by hyphens. Modifier names are alphabetic sequences of characters which the system uses case-insensitively. Following modifiers is a keysym name, which is case-insensitive if it consists of multiple characters, but if the name consists of only a single character, then it is case-sensitive.

You can escape special characters — hyphen, double quote, open angle bracket, close angle bracket, and space — with a backslash, and you can specify a backslash by using two contiguously. You can use angle brackets to enclose a keysym name with many special characters in it. Between angle brackets appearing in a keysym name position, there are only two special characters, the closing angle bracket and backslash.

18.1.2. Interface

All of the following routines and variables are exported from the "EXTENSIONS" ("EXT") package.

define-keysym *keysym preferred-name* **&rest** *other-names* [*Function*]

This function establishes a mapping from *preferred-name* to *keysym* for purposes of **#k** syntax. *Other-names* also map to *keysym*, but the system uses *preferred-name* when printing key-events. The names are case-insensitive simple-strings; however, if the string contains a single character, then it is used case-sensitively. Redefining a keysym or re-using names has undefined effects.

You can use this to define unused keysyms, but primarily this defines keysyms defined in the *X Window System Protocol, MIT X Consortium Standard, X Version 11, Release 4*. **translate-key-event** uses this knowledge to determine what keysyms are modifier keysyms and what keysym stand for alphabetic key-events.

define-mouse-keysym *button keysym name shifted-bit event-key* [*Function*]

This function defines *keysym* named *name* for key-events representing the X *button* cross the X *event-key* (**:button-press** or **:button-release**). *Shifted-bit* is a defined modifier name that **translate-mouse-key-event** sets in the key-event it returns whenever the X shift bit is set in an incoming event.

Note, by default, there are distinct keysyms for each button distinguishing whether the user pressed or released the button.

Keysym should be an one unspecified in *X Window System Protocol, MIT X Consortium Standard, X Version 11, Release 4.*

name-keysym *name* [*Function*]

This function returns the keysym named *name*. If *name* is unknown, this returns **nil**.

keysym-names *keysym* [*Function*]

This function returns the list of all names for *keysym*. If *keysym* is undefined, this returns **nil**.

keysym-preferred-name *keysym* [*Function*]

This returns the preferred name for *keysym*, how it is typically printed. If *keysym* is undefined, this returns **nil**.

define-key-event-modifier *long-name short-name* [*Function*]

This establishes *long-name* and *short-name* as modifier names for purposes of specifying key-events in **#k** syntax. The names are case-insensitive simple-strings. If either name is already defined, this signals an error.

The system defines the following default modifiers (first the long name, then the short name):

- **"Hyper"**, **"H"**
- **"Super"**, **"S"**
- **"Meta"**, **"M"**
- **"Control"**, **"C"**
- **"Shift"**, **"Shift"**
- **"Lock"**, **"Lock"**

`*all-modifier-names*` [*Variable*]

This variable holds all the defined modifier names.

`define-clx-modifier` *clx-mask modifier-name* [*Function*]

This function establishes a mapping from *clx-mask* to a defined key-event *modifier-name*. **`translate-key-event`** and **`translate-mouse-key-event`** can only return key-events with bits defined by this routine.

The system defines the following default mappings between CLX modifiers and key-event modifiers:

- **`(xlib:make-state-mask :mod-1) --> "Meta"`**
- **`(xlib:make-state-mask :control) --> "Control"`**
- **`(xlib:make-state-mask :lock) --> "Lock"`**
- **`(xlib:make-state-mask :shift) --> "Shift"`**

`make-key-event-bits &rest` *modifier-names* [*Function*]

This function returns bits suitable for **`make-key-event`** from the supplied *modifier-names*. If any name is undefined, this signals an error.

`key-event-modifier-mask` *modifier-name* [*Function*]

This function returns a mask for *modifier-name*. This mask is suitable for use with **`key-event-bits`**. If *modifier-name* is undefined, this signals an error.

`key-event-bits-modifiers` *bits* [*Function*]

This returns a list of key-event modifier names, one for each modifier set in *bits*.

`translate-key-event` *display scan-code bits* [*Function*]

This function translates the X *scan-code* and X *bits* to a key-event. First this maps *scan-code* to an X keysym using **`xlib:keycode->keysym`** looking at *bits* and supplying index as **`1`** if the X shift bit is on, **`0`** otherwise.

If the resulting keysym is undefined, and it is not a modifier keysym, then this signals an error. If the keysym is a modifier key, then this returns **`nil`**.

If these conditions are satisfied

- The keysym is defined.
- The X shift bit is off.
- The X lock bit is on.
- The X keysym represents a lowercase letter.

then this maps the *scan-code* again supplying index as **`1`** this time, treating the X lock bit as a caps-lock bit. If this results in an undefined keysym, this signals an error. Otherwise, this makes a key-event with the keysym and bits formed by mapping the X bits to key-event bits.

Otherwise, this makes a key-event with the keysym and bits formed by mapping the X bits to key-event bits.

`translate-mouse-key-event` *scan-code bits event-key* [*Function*]

This function translates the X button code, *scan-code*, and modifier bits, *bits*, for the X *event-key* into a key-event. See **`define-mouse-keysym`**.

make-key-event *object bits* [*Function*]

This function returns a key-event described by *object* with *bits*. *Object* is one of keysym, string, or key-event. When *object* is a key-event, this uses **key-event-keysym**. You can form *bits* with **make-key-event-bits** or **key-event-modifier-mask**.

key-event-p *object* [*Function*]

This function returns whether *object* is a key-event.

key-event-bits *key-event* [*Function*]

This function returns the bits field of a *key-event*.

key-event-keysym *key-event* [*Function*]

This function returns the keysym field of a *key-event*.

char-key-event *character* [*Function*]

This function returns the key-event associated with *character*. You can associate a key-event with a character by **setf**'ing this form.

key-event-char *key-event* [*Function*]

This function returns the character associated with *key-event*. You can associate a character with a key-event by **setf**'ing this form. The system defaultly translates key-events in some implementation dependent way for text insertion; for example, under an ASCII system, the key-event **#k"C-h"**, as well as **#k"backspace"** would map to the Common Lisp character that causes a backspace.

key-event-bit-p *key-event bit-name* [*Function*]

This function returns whether *key-event* has the bit set named by *bit-name*. This signals an error if *bit-name* is undefined.

do-alpha-key-events (*var kind* **&optional** *result***)** {*form*}* [*Macro*]

This macro evaluates each *form* with *var* bound to a key-event representing an alphabetic character. *Kind* is one of **:lower**, **:upper**, or **:both**, and this binds *var* to each key-event in order as specified in *X Window System Protocol, MIT X Consortium Standard, X Version 11, Release 4*. When **:both** is specified, this processes lowercase letters first.

print-pretty-key *key* **&optional** *stream long-names-p* [*Function*]

This prints *key*, a key-event or vector of key-events, in a user-expected fashion to *stream*. *Long-names-p* indicates whether modifiers should print with their long or short name. *Stream* defaults to ***standard-output***.

print-pretty-key-event *key-event* **&optional** *stream long-names-p* [*Function*]

This prints *key-event* to *stream* in a user-expected fashion. *Long-names-p* indicates whether modifier names should appear using the long name or short name. *Stream* defaults to ***standard-output***.

18.2. CLX Interface

18.2.1. Graphics Window Hooks

This section describes a few hooks used by Hemlock's internals to handle graphics windows that manifest Hemlock windows. Some heavy users of Hemlock as a tool have needed these in the past, but typically functions that replace the default values of these hooks must be written in the "**`HEMLOCK-INTERNALS`**" package. All of these symbols are internal to this package.

If you need this level of control for your application, consult the current implementation for code fragments that will be useful in correctly writing your own window hook functions.

`*create-window-hook*` [*Variable*]

This holds a function that Hemlock calls when **`make-window`** executes under CLX. Hemlock passes the CLX display and the following arguments from **`make-window`**: starting mark, ask-user, x, y, width, height, and modelinep. The function returns a CLX window or nil indicating one could not be made.

`*delete-window-hook*` [*Variable*]

This holds a function that Hemlock calls when **`delete-window`** executes under CLX. Hemlock passes the CLX window and the Hemlock window to this function.

`*random-typeout-hook*` [*Variable*]

This holds a function that Hemlock calls when random typeout occurs under CLX. Hemlock passes it a Hemlock device, a pre-existing CLX window or **`nil`**, and the number of pixels needed to display the number of lines requested in the **`with-pop-up-display`** form. It should return a window, and if a new window is created, then a CLX gcontext must be the second value.

`*create-initial-windows-hook*` [*Variable*]

This holds a function that Hemlock calls when it initializes the screen manager and makes the first windows, typically windows for the Main and Echo Area buffers. Hemlock passes the function a Hemlock device.

18.2.2. Entering and Leaving Windows

Enter Window Hook [Hemlock *Variable*]

When the mouse enters an editor window, Hemlock invokes the functions in this hook. These functions take a Hemlock window as an argument.

Exit Window Hook [Hemlock *Variable*]

When the mouse exits an editor window, Hemlock invokes the functions in this hook. These functions take a Hemlock window as an argument.

18.2.3. How to Lose Up-Events

Often the only useful activity user's design for the mouse is to click on something. Hemlock sees a character representing the down event, but what do you do with the up event character that you know must follow? Having the command eat it would be tasteless, and would inhibit later customizations that make use of it, possibly adding on to the down click command's functionality. Bind the corresponding up character to the command described here.

Do Nothing [*Command*]

This does nothing as many times as you tell it.

18.3. Slave Lisps

Some implementations of Hemlock feature the ability to manage multiple slave Lisps, each connected to one editor Lisp. The routines discussed here spawn slaves, send evaluation and compilation requests, return the current server, etc. This is very powerful because without it you can lose your editing state when code you are developing causes a fatal error in Lisp.

The routines described in this section are best suited for creating editor commands that interact with slave Lisps, but in the past users implemented several independent Lisps as nodes communicating via these functions. There is a better level on which to write such code that avoids the extra effort these routines take for the editor's sake. See the *CMU Common Lisp User's Manual* for the **`remote`** and **`wire`** packages.

18.3.1. The Current Slave

There is a slave-information structure that these return which is suitable for passing to the routines described in the following subsections.

`create-slave &optional` *name* [*Function*]

This creates a slave that tries to connect to the editor. When the slave connects to the editor, this returns a slave-information structure, and the interactive buffer is the buffer named *name*. This generates a name if *name* is **`nil`**. In case the slave never connects, this will eventually timeout and signal an editor-error.

`get-current-eval-server &optional` *errorp* [*Function*]
Current Eval Server [Hemlock *Variable*]

This returns the server-information for the Current Eval Server after making sure it is valid. Of course, a slave Lisp can die at anytime. If this variable is **`nil`**, and *errorp* is non-**`nil`**, then this signals an editor-error; otherwise, it tries to make a new slave. If there is no current eval server, then this tries to make a new slave, prompting the user based on a few variables (see the *Hemlock User's Manual*).

`get-current-compile-server` [*Function*]
Current Compile Server [Hemlock *Variable*]

This returns the server-information for the Current Compile Server after making sure it is valid. This may return nil. Since multiple slaves may exist, it is convenient to use one for developing code and one for compiling files. The compilation commands that use slave Lisps prefer to use the current compile server but will fall back on the current eval server when necessary. Typically, users only have separate compile servers when the slave Lisp can live on a separate workstation to save cycles on the editor machine, and the Hemlock commands only use this for compiling files.

18.3.2. Asynchronous Operation Queuing

The routines in this section queue requests with an eval server. Requests are always satisfied in order, but these do not wait for notification that the operation actually happened. Because of this, the user can continue editing while his evaluation or compilation occurs. Note, these usually execute in the slave immediately, but if the interactive buffer connected to the slave is waiting for a form to return a value, the operation requested must wait until the slave is free again.

string-eval *string* **&key :server :package :context** [*Function*]
region-eval *region* **&key :server :package :context** [*Function*]
region-compile *region* **&key :server :package** [*Function*]

string-eval queues the evaluation of the form read from *string* on eval server *server*. *Server* defaults to the result of **get-current-server**, and *string* is a simple-string. The evaluation occurs with ***package*** bound in the slave to the package named by *package*, which defaults to Current Package or the empty string; the empty string indicates that the slave should evaluate the form in its current package. The slave reads the form in *string* within this context as well. *Context* is a string to use when reporting start and end notifications in the Echo Area buffer; it defaults to the concatenation of **"evaluation of "** and *string*.

region-eval is the same as **string-eval**, but *context* defaults differently. If the user leaves this unsupplied, then it becomes a string involving part of the first line of region.

region-compile is the same as the above. *Server* defaults the same; it does not default to **get-current-compile-server** since this compiles the region into the slave Lisp's environment, to affect what you are currently working on.

file-compile *file* **&key :output-file :error-file :load :server :package** [*Function*]
Remote Compile File (initial value **nil**) [Hemlock *Variable*]

This compiles *file* in a slave Lisp. When *output-file* is **t** (the default), this uses a temporary output file that is publicly writable in case the client is on another machine, which allows for file systems that do not permit remote write access. This renames the temporary file to the appropriate binary name or deletes it after compilation. Setting Remote Compile File to **nil**, inhibits this. If *output-file* is non-**nil** and not **t**, then it is the name of the binary file to write. The compilation occurs with ***package*** bound in the slave to the package named by *package*, which defaults to Current Package or the empty string; the empty string indicates that the slave should evaluate the form in its current package. *Error-file* is the file in which to record compiler output, and a **nil** value inhibits this file's creation. *Load* indicates whether to load the resulting binary file, defaults to **nil**. *Server* defaults to **get-current-compile-server**, but if this returns nil, then *server* defaults to **get-current-server**.

18.3.3. Synchronous Operation Queuing

The routines in this section queue requests with an eval server and wait for confirmation that the evaluation actually occurred. Because of this, the user cannot continue editing while the slave executes the request. Note, these usually execute in the slave immediately, but if the interactive buffer connected to the slave is waiting for a form to return a value, the operation requested must wait until the slave is free again.

eval-form-in-server *server-info string* **&optional** *package* [*Function*]

This function queues the evaluation of a form in the server associated with *server-info* and waits for the results. The server **read**'s the form from *string* with ***package*** bound to the package named by *package*. This returns the results from the slave Lisp in a list of string values. You can **read** from the strings or simply display them depending on the **print**'ing of the evaluation results.

Package defaults to Current Package. If this is **nil**, the server uses the value of ***package*** in the server.

While the slave executes the form, it binds ***terminal-io*** to a stream that signals errors when read from and dumps output to a bit-bucket. This prevents the editor and slave from dead locking by waiting for each other to reply.

eval-form-in-server-1 *server-info string* **&optional** *package* [*Function*]

This function calls **eval-form-in-server** and **read**'s the result in the first string it returns. This result must be **read**'able in the editor's Lisp.

18.4. Spelling

Hemlock supports spelling checking and correcting commands based on the ITS Ispell dictionary. These commands use the following routines which include adding and deleting entries, reading the Ispell dictionary in a compiled binary format, reading user dictionary files in a text format, and checking and correcting possible spellings.

spell:maybe-read-spell-dictionary [*Function*]

This reads the default binary Ispell dictionary. Users must call this before the following routines will work.

spell:spell-read-dictionary *filename* [*Function*]

This adds entries to the dictionary from the lines in the file *filename*. Dictionary files contain line oriented records like the following:

```
entry1/flag1/flag2
entry2
entry3/flag1
```

The flags are the Ispell flags indicating which endings are appropriate for the given entry root, but these are unnecessary for user dictionary files. You can consult Ispell documentation if you want to know more about them.

spell:spell-add-entry *line* **&optional** *word-end* [*Function*]

This takes a line from a dictionary file, and adds the entry described by *line* to the dictionary. *Word-end* defaults to the position of the first slash character or the length of the line. *Line* is destructively modified.

spell:spell-remove-entry *entry* [*Function*]

This removes entry, a simple-string, from the dictionary, so it will be an unknown word. This destructively modifies *entry*. If it is a root word, then all words derived with *entry* and its flags will also be deleted. If *entry* is a word derived from some root word, then the root and any words derived from it remain known words.

spell:correct-spelling *word* [*Function*]

This checks the spelling of *word* and outputs the results. If this finds *word* is correctly spelled due to some appropriate suffix on a root, it generates output indicating this. If this finds *word* as a root entry, it simply outputs that it found *word*. If this cannot find *word* at all, then it outputs possibly correct close spellings. This writes to ***standard-output***, and it calls **maybe-read-spell-dictionary** before attempting any lookups.

spell:spell-try-word *word word-len* [*Function*]
max-entry-length [*Constant*]

This returns an index into the dictionary if it finds *word* or an appropriate root. *Word-len* must be inclusively in the range 2 through **max-entry-length**, and it is the length of *word*. *Word* must be uppercase. This returns a second value indicating whether it found *word* due to a suffix flag, **nil** if *word* is a root entry.

spell:spell-root-word *index* [*Function*]

This returns a copy of the root word at dictionary entry *index*. This index is the same as returned by **spell-try-word**.

spell:spell-collect-close-words *word* [*Function*]

This returns a list of words correctly spelled that are *close* to *word*. *Word* must be uppercase, and its length must be inclusively in the range 2 through **max-entry-length**. Close words are determined by the Ispell rules:

1. Two adjacent letters can be transposed to form a correct spelling.
2. One letter can be changed to form a correct spelling.
3. One letter can be added to form a correct spelling.
4. One letter can be removed to form a correct spelling.

spell:spell-root-flags *index* [*Function*]

This returns a list of suffix flags as capital letters that apply to the dictionary root entry at *index*. This index is the same as returned by **spell-try-word**.

18.5. File Utilities

Some implementations of Hemlock provide extensive directory editing commands, Dired, including a single wildcard feature. An asterisk denotes a wildcard.

dired:copy-file *spec1 spec2* **&key :update :clobber :directory** [*Function*]

This function copies *spec1* to *spec2*. It accepts a single wildcard in the filename portion of the specification, and it accepts directories. This copies files maintaining the source's write date.

If *spec1* and *spec2* are both directories, this recursively copies the files and subdirectory structure of *spec1*; if *spec2* is in the subdirectory structure of *spec1*, the recursion will not descend into it. Use **"/spec1/*"** to copy only the files from *spec1* to directory *spec2*.

If *spec2* is a directory, and *spec1* is a file, then this copies *spec1* into *spec2* with the same **pathname-name**.

When **:update** is non-**nil**, then the copying process only copies files if the source is newer than the destination.

When **:update** and **:clobber** are **nil**, and the destination exists, the copying process stops and asks the user whether the destination should be overwritten.

When the user supplies **:directory**, it is a list of pathnames, directories excluded, and *spec1* is a pattern containing one wildcard. This then copies each of the pathnames whose **pathname-name** matches the pattern. *Spec2* is either a directory or a pathname whose **pathname-name** contains a wildcard.

dired:rename-file *spec1 spec2* **&key :clobber :directory** [*Function*]

This function renames *spec1* to *spec2*. It accepts a single wildcard in the filename portion of the specification, and *spec2* may be a directory with the destination specification resulting in the merging of *spec2* with *spec1*. If **:clobber** is **nil**, and *spec2* exists, then this asks the user to confirm the renaming. When renaming a directory, end the specification without the trailing slash.

When the user supplies **:directory**, it is a list of pathnames, directories excluded, and *spec1* is a pattern containing one wildcard. This then copies each of the pathnames whose **pathname-name** matches the pattern. *Spec2* is either a directory or a pathname whose **pathname-name** contains a wildcard.

dired:delete-file *spec* **&key :recursive :clobber** [*Function*]

This function deletes *spec*. It accepts a single wildcard in the filename portion of the specification, and it asks for confirmation on each file if **:clobber** is **nil**. If **:recursive** is non-**nil**, then *spec* may be a directory to recursively delete the entirety of the directory and its subdirectory structure. An empty directory may be specified without **:recursive** being non-**nil**. Specify directories with the trailing slash.

dired:find-file *name* **&optional** *directory find-all* [*Function*]

This function finds the file with **file-namestring** *name*, recursively looking in *directory*. If *find-all* is non-**nil** (defaults to **nil**), then this continues searching even after finding a first occurrence of file. *Name* may contain a single wildcard, which causes *find-all* to default to **t** instead of **nil**.

dired:make-directory *name* [*Function*]

This function creates the directory with *name*. If it already exists, this signals an error.

dired:pathnames-from-pattern *pattern files* [*Function*]

This function returns a list of pathnames from the list *files* whose **file-namestring**'s match *pattern*. *Pattern* must be a non-empty string and contain only one asterisk. *Files* contains no directories.

dired:*update-default* [*Variable*]
dired:*clobber-default* [*Variable*]
dired:*recursive-default* [*Variable*]

These are the default values for the keyword arguments above with corresponding names. These default to **nil**, **t**, and **nil** respectively.

dired:*report-function* [*Variable*]
dired:*error-function* [*Variable*]
dired:*yesp-function* [*Variable*]

These are the function the above routines call to report progress, signal errors, and prompt for *yes* or *no*. These all take format strings and arguments.

merge-relative-pathnames *pathname default-directory* [*Function*]

This function merges *pathname* with *default-directory*. If *pathname* is not absolute, this assumes it is relative to *default-directory*. The result is always a directory pathname.

directoryp *pathname* [*Function*]

This function returns whether *pathname* names a directory: it has no name and no type fields.

18.6. Beeping

hemlock-beep [*Function*]

Hemlock binds **system:*beep-function*** to this function to beep the device. It is different for different devices.

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

Bell Style determines what **`*hemlock-beep*`** does in Hemlock under CLX. 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.

Index

Index

Table of Contents