1987

Hemlock command implementor's manual

Rob MacLachlan
*Carnegie Mellon University*

Follow this and additional works at: [http://repository.cmu.edu/compsci](http://repository.cmu.edu/compsci)
NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.
Hemlock Command Implementor's Manual

Rob MacLachlan

September 1987

CMU-CS-87-159.

Abstract

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

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

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

Table of Contents

1. Introduction
2. Representation of Text
   2.1. Lines
   2.2. Marks
      2.2.1. Kinds of Marks
      2.2.2. Mark Functions
      2.2.3. Making Marks
      2.2.4. Moving Marks
   2.3. Regions
      2.3.1. Region Functions
3. Buffers
   3.1. The Current Buffer
   3.2. Buffer Functions
4. Predicates
   4.1. Type Predicates
   4.2. Text Predicates
5. Doing Stuff and Going Places
   5.1. Altering Text
   5.2. Searching and Replacing
6. The Current Environment
   6.1. Different Scopes
   6.2. Shadowing
7. Hemlock Variables
   7.1. Variable Names
   7.2. Variable Functions
   7.3. Hooks
8. Commands
   8.1. Introduction
      8.1.1. Defining Commands
      8.1.2. Command Documentation
   8.2. The Command Interpreter
      8.2.1. Binding Commands to Keys
      8.2.2. Key Translation
      8.2.3. Transparent Key Bindings
   8.3. Command Types
   8.4. Command Arguments
      8.4.1. The Prefix Argument
      8.4.2. Lisp Arguments
   8.5. Recursive Edits
9. Modes
   9.1. Mode Hooks
   9.2. Major and Minor Modes
   9.3. Mode Functions
10. Character Attributes
   10.1. Introduction
   10.2. Character Attribute Names
   10.3. Character Attribute Functions
   10.4. Character Attribute Hooks
11. Controlling the Display 35
  11.1. Windows 35
  11.2. The Current Window 35
  11.3. Modelines 35
  11.4. Window Functions 36
  11.5. Cursor Positions 37
  11.6. Redisplay 38

12. Logical Characters 39
  12.1. What a Logical Character is 39
  12.2. Logical Character Functions 39
  12.3. Standard Logical Characters 40

13. The Echo Area 41
  13.1. Echo Area Clearing 41
  13.2. Echo Area Functions 41
  13.3. Prompting Functions 42
  13.4. Control of Parsing Behavior 44
  13.5. Defining New Prompting Functions 45
  13.6. Standard Echo Area Commands 46

14. Hemlock's Lisp Environment 47
  14.1. Entering and Leaving the Editor 47
  14.2. Miscellaneous 47
  14.3. Keyboard Input 47
  14.4. Hemlock Streams 48
  14.5. Interface to the Error System 49
  14.6. File Reading and Writing 49

15. Utilities 51
  15.1. String-table Functions 51
  15.2. Manipulating Ring Buffers 52

Index 53
Index 55
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 8.
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.

(line-string line)  [Function]
Given a line, returns as a simple string the characters in the line, turned. setf can be used 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, returns line-previous 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]
Returns the buffer which contains this line. Note that a line may not be associated with any buffer, in which case line-buffer returns nil.

(line-length line)  [Function]
Returns the number of characters in the line. This does not include the newline character at the end.

(line-character line index)  [Function]
Return 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 then a newline is returned.

(line-plist line)  [Function]
Returns the property-list for line. setf, getf, putf and remf can be used to change properties. This is usually used in conjunction with line-signature to cache information about the line's contents.

(line-signature line)  [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 eq to any previous value. Note that the signature may change even when the text hasn't been modified, but this probably won't 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.

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 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

`mark-line mark` [Function]
Returns the line that `mark` points to. 

`mark-charpos mark` [Function]
Returns the character position the `mark` points to.

`mark-kind mark` [Function]
Returns one of :right-inserting, :left-inserting or :temporary depending on the mark's kind. A corresponding `setq` form changes the mark's kind.

`previous-character mark` [Function]
`next-character mark` [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 `setq`.

2.2.3. Making Marks

`mark line charpos &optional kind` [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]
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]
Deletes the mark. This should be done to any mark which may be permanent when it is no longer needed.

with-mark (([mark pos [kind]])*) [form]*

[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.

move-to-position mark charpos &optional line

[Function]
Changes the mark to point to the given character position on the line line. line defaults to the line the mark is currently on.

move-mark mark new-position

[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]
Changes mark to point to the beginning or the end of line and returns it. line defaults to the line that mark is currently on.

buffer-start mark &optional buffer

[Function]
buffer-end mark &optional buffer

[Function]
Change mark to point to the beginning or end of buffer, which defaults to the buffer mark currently points into. If buffer is not supplied then it is an error for mark not to point into some buffer.

mark-before mark

[Function]
mark-after mark

[Function]
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]
Changes mark to point n characters after (n before if n is negative) the current position. If there aren't n characters after (before) the mark, then nil is returned and mark is not modified.

line-offset mark n &optional charpos

[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

\[
\min (\text{line-length } \text{resulting-line}) (\text{mark-charpos } \text{mark})
\]

if charpos is unspecified, or

\[
\min (\text{line-length } \text{resulting-line}) \text{charpos}
\]

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

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.

make-empty-region

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

Returns a region containing a copy of the text in the specified region.

region-to-string region

String-to-region string

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

line-to-region line

Returns a region containing all the characters on line. The first mark is right-inserting and the last is left-inserting.

region-start region

region-end region

Returns the start or end mark of region.

region-bounds region

Return as multiple-values the starting and ending marks of region.

set-region-bounds region start end

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

count-lines region

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

Returns the number of characters in a given region. The line breaks are counted as one character.
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.
5. A write protect flag.
6. Some variables (page 19).
7. Some key bindings (page 24).
8. Some collection of modes (page 29).
9. Some windows in which it is displayed (page 35).

3.1. The Current Buffer

**current-buffer**

*Function*

Set Buffer Hook

After Set Buffer Hook

current-buffer returns the current buffer object. Usually this is the buffer that current-window (page 35) is displaying. This value may be changed with setf, in which case Set Buffer Hook is invoked beforehand with the new value. After the buffer is changed, After Set Buffer Hook is invoked 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.

**buffer-list**

*Variable*

Holds a list of all the buffer objects made with make-buffer.

**buffer-names**

*Variable*

Holds a string-table (page 51) of all the names of the buffers in *buffer-list*. The values of the entries are the corresponding buffer objects.
3.2. Buffer Functions

`make-buffer name &optional modes`  
Make Buffer Hook  

`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 29).

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.

`buffer-name buffer`  
Buffer Name Hook  

`buffer-name` returns the name of the given `buffer`, a string. The corresponding `setf` form sets the buffer name. If an attempt is made to set the buffer name to one that already exists then no renaming is done and `nil` is returned. The hook `Buffer Name Hook` is invoked with the buffer and the new name when the name is changed.

`buffer-region buffer`  

Returns the buffer's region. This can be set with `setf`.

`buffer-pathname buffer`  
Buffer Pathname Hook  

`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`  

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`  

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-writable buffer`  

Returns `t` if the `buffer` can be altered, `nil` if it can't. There is a `setf` form to change this value. If a buffer is not writable, then any attempt to modify text in the buffer will result in an error.

`buffer-modified buffer`  
Buffer Modified Hook  

`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` form to change this value.

`Buffer Modified Hook` is invoked with the buffer whenever the value of the modified flag changes.
**buffer-variables** buffer

Returns a string-table (page 51) containing the names of the buffer's local variables. See chapter 7.

**buffer-modes** buffer

Returns the list of the names of the modes active in buffer. The major mode is first, followed by any minor modes. See chapter 9.

**buffer-windows** buffer

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 35 for a discussion of windows.

**delete-buffer** buffer

Delete Buffer Hook [Hemlock Variable]

delete-buffer removes buffer from *buffer-list* (page 9) and its name from *buffer-names* (page 9). Before the buffer is deleted the hook Delete Buffer Hook is invoked with the buffer.
Chapter 4
Predicates

4.1. Type Predicates

The following are implemented as structures and thus have type predicates defined: line, mark, region, buffer, window, string-table, ring, command and search-pattern.

4.2. Text Predicates

[start-line-p mark
Returns t if the mark points before the first character in a line, nil otherwise.
[end-line-p mark
Returns t if the mark points after the last character in a line and before the newline, nil otherwise.
[empty-line-p mark
Return t of the line which mark points to contains no characters.
[blank-line-p line
Returns t if line contains only characters with a Whitespace attribute of 1. See chapter 10 for discussion of character attributes.
[blank-before-p mark
[blank-after-p mark
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
Returns t if mark1 and mark2 point to the same line, or nil otherwise. i.e.: (same-line-p a b) <=t (eq (mark-line a) (mark-line b))
[mark< mark1 mark2
[mark<= mark1 mark2
[mark= mark1 mark2
[mark/= mark1 mark2
[mark>= mark1 mark2
[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< linel line2` [Function]
- `line<= linel line2` [Function]
- `line>= linel line2` [Function]
- `line> linel 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 linel 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`. 
Chapter 5

Doing Stuff and Going Places

5.1. 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
insert-string mark string
insert-region mark region

Inserts a character, string or region at mark.

ninsert-region mark region

Line 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.

delete-characters mark n

delete-region region

Deletes the region. This is faster than delete-and-save-region (below) because no lines are copied.

delete-and-save-region region

Deletes the region, and returns a region containing the original region's text.

filter-region function region

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:
5.2. Searching and Replacing

Before using any of these functions to do a character search, look at character attributes (page 31). They provide a facility similar to the syntax table in real EMACS. Syntax tables are a powerful, general, efficient, and otherwise generally winning way of dealing with what characters do what in which mode. Character attributes in Hemlock are even more general way of attacking this problem.

**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. Bits and font are always ignored.

**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 in 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.

**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 6

The Current Environment

6.1. Different Scopes

In Hemlock the values of variables (page 19), key-bindings (page 24) and character-attributes (page 31) 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.

6.2. Shadowing

It is possible for there to be a conflict between different values for the same thing in different scopes. For example, there might be 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 7

Hemlock Variables

Hemlock implements a system of variables separate from the normal Lisp variables; this is done for the following reasons.

1. Hemlock has different scope rules which are useful in an editor. Hemlock variables can be local to a buffer (page 9) or a mode (page 29).
2. Hemlock variables have hooks (page 21), functions which are called when the variable is set.
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.

7.1. 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 will not work.

*global-variable-names* [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.

7.2. Variable Functions

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

defhvar string-name documentation &key :mode :buffer :hooks :value [Function]
Defines a Hemlock variable. An error will be signaled if a reference is made to a variable which is not defined.

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, likewise if mode is supplied it is local to that mode. If neither is supplied it is global.
:hooks :value The initial hook-list and value for the variable, which default to nil.
If a variable with the same name is already declared in the same place then its hooks and value are set to
the value of hooks and value when these keywords are supplied.

variable-value name &optional kind where
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 the variable name.
:mode Return value for name in the mode named where.
:buffer Return the value for name in the buffer where.
When set with setf, the value of the specified variable is set and the functions in its hook list are called with the values for name, kind, where and the new value.

variable-documentation name &optional kind where
variable-hooks name &optional kind where
variable-name name &optional kind where
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
This function converts a string into the corresponding variable symbol name. string need not be the name of an actual Hemlock variable.

value name
setv name new-value
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)*
This macro is very similar to let in effect: within its scope each of the Hemlock variables var have the respective values, 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
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
Delete Variable Hook
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.
7.3. Hooks

Hemlock actions often have hooks associated with them, which are lists of functions to be called before that action is performed. Each variable and mode has such a hook, and the ways to manipulate these object-specific hooks are described with the rest of the actions defined on these objects. Many events that affect editor state also will call functions in a hook list; these hooks are described along with the functions that invoke them.

A hook function may be specified either as a symbol with a function definition or a function, but it is recommended to use symbols, since this results in better behavior if the hook function is redefined.

\begin{verbatim}
add-hook place hook-fun                              [Macro]
remove-hook place hook-fun                           [Macro]

These macros add or remove a hook function in some place. If place is a symbol then it is interpreted as a Hemlock variable, it is taken to be a generalized variable.

invoke-hook name &rest args                         [Function]

Call all the functions in the list which is the value of the Hemlock variable name. An error will be signalled if no such variable is defined.
\end{verbatim}
Chapter 8

Commands

8.1. 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 can also be called from Lisp.

*command-names* [Variable]

Holds a string-table (page 51) associating command names to command objects. Whenever a new command is defined it is entered in this table.

8.1.1. Defining Commands

`def command` [command-name | (command-name function-name)] lambda-list

`command-doc` function-doc [form] [Macro]

Defines a command named `name`, `def command` creates a function to implement the command from the `lambda-list` and `forms` supplied. If not specified, the function name is made from 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 23), 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.

`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`. 


8.1.2. 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.

8.2. The Command Interpreter

The command interpreter is a function which reads keystrokes from the keyboard and dispatches to different commands on the basis of what is typed. When the command interpreter calls a command, it is said in invoke it. The command interpreter also provides several facilities for communication between sequential commands and does various house cleaning operations.

*invoke-hook* [Variable]

This variable contains a function which is called by the command interpreter when it wants to invoke a command. The function is passed 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.

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 characters read. Commands may call the command interpreter recursively using the function recursive-edit (page 27).

8.2.1. 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 keystrokes. A sequence of keystrokes is called a key, and is represented by a single character or a sequence (list or vector) of characters.

The set of key bindings in effect at any given time is determined by the current environment (page 17), since key bindings may be local to a mode or buffer. When the command interpreter tries to find the binding for a key it checks first to see 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 29), 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.

command-char-code-limit [Constant]

command-char-bits-limit [Constant]

Hemlock implementation is not required to support entirely arbitrary characters in key bindings: command-char-code-limit is the upper bound on character codes, and command-char-bits-limit is the limit for bits. These constants are analogous to the COMMON LISP constants char-code-limit and char-bits-limit, and will be less than or equal to them.
Bits not supported and font are ignored. Note that no attempt is made to define some virtual character set in which bindings can be specified in an implementation independent fashion; key bindings should be set up in file that contains nothing else so that they may be easily changed for different implementations.

bind-key name key optional kind where
Make key be bound to the command name in some environment. 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.
If the specified 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. Normally global and mode bindings are made only at load time.

command-bindings command
Returns a list of the places where command is bound. A place is specified as a list of the key vector, the kind of binding, and then either the mode of buffer the binding is local to, or nil if it is a global binding.

delete-key-binding key optional kind where
Removes the binding of key in some place. 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, if kind is :buffer then where is the buffer. An error will be signaled if key is not bound.

get-command key optional kind where
Returns the command bound to key; if key is not bound return nil. If the sequence given is a prefix and not a unique key then the keyword :prefix is returned. There are four cases of kind:

:current Return the current binding of key using the current buffer's search list. This is the default. 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.
:mode Return the binding of key in the mode named where.
:buffer Return the binding of key local to the buffer where.

map-bindings function kind optional where
This function maps over the key-bindings in some place. For each binding function is passed the key bound and the command bound to it. kind and where are as in get-command, except that :current is not available. The key is not guaranteed to remain valid after a given iteration.

8.2.2. 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, key translation is done when a specified character sequence appears in a key.

In a substitution translation, the matched subsequence is replaced with another character sequence. Key translation is not recursively applied to the substituted characters.

In a bit-prefix translation, the matched subsequence is removed, and specified bits are set in the next character in the key.
If the key being translated ends in a prefix of some translation, or if there is no character following a bit-prefix translation, then the matched characters are not translated. If there is a binding for this partially untranslated key, then the command interpreter will invoke that command, otherwise it will wait for more characters to be typed.

`key-translation key`

Return the key translation for `key`, or `nil` if there is none. If `key` is a prefix of a translation, then `:prefix` is returned. Whenever `key` appears as a subsequence of a key argument to the binding manipulation functions, that portion will be replaced with the translation. A key translation may also be a list `(:bits (bit-name)*)`. In this case, the named bits will be set in the next character in the key being translated.

8.2.3. 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 30) determines whether the key bindings in a mode are transparent or not.

8.3. Command Types

In many editors the exact behavior of a command depends on what kind of commands have been invoked before it. Hemlock provides a mechanism to support this: The concept of command type.

`last-command-type`

Return the command type of the last command invoked. If set with `setf`, the supplied value becomes the value of `last-command-type` until the next command completes or it is reset. If the previous command did not bother to set the `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.

8.4. 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.

8.4.1. 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

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.

8.4.2. 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.

8.5. Recursive Edits

**use-buffer** buffer [form]*

[Macro]

The effect of this is similar to that which would be obtained by setting the current-buffer to `buffer` during the evaluation of `forms`. There are, however, 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 the current buffer to `buffer` and back.

**recursive-edit &optional handle-abort**

[Function]

Enter Recursive Edit Hook

`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.

**exit-recursive-edit &optional values-list**

[Function]

Exit Recursive Edit Hook

`exit-recursive-edit` exits a recursive edit, returning all the things in `values-list`, which defaults to `nil`, as multiple-values. After the command interpreter is exited the hook Exit Recursive Edit Hook is invoked. If no recursive edit is in progress then `ed` returns with the values.

**abort-recursive-edit &rest args**

[Function]

Abort Recursive Edit Hook

`abort-recursive-edit` causes a recursive edit to terminate with the error given. The arguments are the same as `editor-error` (page 49). Abort Recursive Edit Hook is invoked before the recursive edit is aborted with the `editor-error` arguments. If no recursive edit is in progress then Hemlock returns with a string representing the message, if any, or `NIL` otherwise.
Chapter 9

Modes

A mode is a collection of Hemlock values which may be present in the current environment (page 17) 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.

9.1. 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.

9.2. 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"))

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

*mode-names*

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

9.3. Mode Functions
defmode name &key :setup-function :cleanup-function :major-p :precedence :transparent-p

This function defines a new mode named name, and enters it in *mode-names* (page 29). 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.

buffer-major-mode buffer
Buffer Major Mode Hook
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
Buffer Minor Mode Hook
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
Returns the string-table of mode local variables.

mode-major-p name
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 10

Character Attributes

10.1. 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. Character attributes are used 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 16).

character-attribute-char-code-limit

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

10.2. 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*

Whenever a character attribute is defined, its name is entered in this string table (page 51), with the corresponding keyword as the value.
10.3. Character Attribute Functions

```
defattribute name documentation optional type initial-value
Make Character Attribute Hook
```

*Function*

`defattribute` defines a new character attribute with string name `name`. `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.

The hook, Make Character Attribute Hook, is invoked with the same arguments after the attribute is created.

```
character-attribute-name attribute
character-attribute-documentation attribute
```

*Function*

`character-attribute-name` attribute
`character-attribute-documentation` attribute

Return the name or documentation for `attribute`.

```
character-attribute attribute character
Character Attribute Hook
```

*Function*

`character-attribute` attribute character

Character Attribute Hook

Function returns the value of `attribute` for `character`. An error will be signaled if `attribute` is not defined.

`setf` can be used to set a character's attributes. The hook Character Attribute Hook, is invoked with the same arguments before the change is made.

If `character` is `nil`, then the value of the attribute for the beginning or end of the 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*

Returns `t` if `symbol` is the name of a character attribute, `nil` otherwise.

```
shadow-attribute attribute character value mode
Shadow Attribute Hook
```

*Function*

`shadow-attribute` attribute character value mode

Shadow Attribute Hook

Makes have `attribute` have value `value` when in 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
Unshadow Attribute Hook
```

*Function*

`unshadow-attribute` attribute character mode

Unshadow Attribute Hook

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

```
find-attribute mark attribute &optional test
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`. `test` is passed one argument, the value of `attribute` for the character to be tested. If the test succeeds then `mark` is modified to point before (after for `reverse-find-attribute`) the character which satisfied the test, if no character is found which satisfies the test then `nil` is returned and
mark is unmodified. test defaults to not zerop. It is not guaranteed that the test will be applied in any particular fashion, so it should have no side effects and depend only on its argument.

10.4. Character Attribute Hooks

It is often useful to use the character attribute mechanism to 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
Return the current hook list for attribute. This may be set with setf. The add-hook and remove-hook (page 21) macros should be used to manipulate these lists.
Chapter 11

Controlling the Display

11.1. 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.

11.2. 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.

11.3. Modelines

A window may have a `modeline`; a line of text which is displayed across the bottom of a window to indicate status information, typically related to the buffer displayed.

A modeline is specified by two things, a string and a function. The string is a `format` control string to generate the modeline, and the function is a function which when called with the window as an argument returns multiple-values to be used as the `format` arguments.

`window-modeline-string` `window` [Function]

`window-modeline-function` `window` [Function]

Return the modeline string or function for `window`. These may be changed with `setf`.

`update-window-modeline` `window` [Function]

This function indicates to Hemlock that at some point in the near future it should recompute the modeline for `window`. In order for changes to appear in the modeline, this function must be called. Usually this is done by defining hooks for the things displayed in the modeline which do this.
make-window mark &key :modeline-string :modeline-function :ask-user  [Function]
Default Modeline String (initial value "Hemlock -A -:[~A: ~]:[-A--:[~; *-]]")  [Hemlock Variable]
Default Modeline Function (initial value default-modeline-function-function) [Hemlock Variable]
Make Window Hook [Hemlock Variable]

make-window returns a window displaying text starting at mark, which must point into a buffer.

modeline-string and modeline-function specify the modeline for the window. If modeline-string is nil then the window has no modeline. Default Modeline Function and Default Modeline String are the default values for these arguments.

If ask-user is t, then the user will be prompted for the dimensions when the device supports prompting. If ask-user is false, then prompting will never be done. Non-null values other than t may have device-dependent meanings.

Make Window Hook is invoked with the new window.

*window-list* [Variable]
Holds a list of all the window objects made with make-window (page 36).

delete-window window  [Function]
Delete Window Hook [Hemlock Variable]
delete-window makes window go away, first invoking Delete Window Hook with the hapless window.

11.4. Window Functions

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-point window  [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 then the window point is the position that the buffer point will be moved to when the window is made current.
center-window window mark
Attempts to adjust window’s display start so that mark is vertically centered within the window.

scroll-window window n
Scroll 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
Returns t if either the character before or the character after mark is being displayed in window, or nil otherwise.

window-height window
window-width window
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
previous-window window
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.)

11.5. 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
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
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
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. nil is returned if no information is available.

mark-column mark
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 \textit{mark} \textit{column} \textit{optional line} \hfill [Function]

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

\textbf{show-mark} \textit{mark} \textit{window} \textit{time} \hfill [Function]

Highlights the position of \textit{mark} within \textit{window} for \textit{time} seconds, possibly by moving the cursor there. The wait may be aborted if there is pending input. If \textit{mark} is not positioned within the text displayed by \textit{window} then return \texttt{nil}, otherwise return \texttt{t}.

\section{11.6. Redisplay}

Redisplay is the process by which the editor translates changes in the internal representation of text into changes on the screen. Ideally this process should find the minimal transformation of the screen which would bring it in correspondence with the text in order to maximize the speed at which it is done.

\textbf{redisplay} \hfill [Function]

Cause the redisplay process to be invoked. This is usually done by the command interpreter after the completion of each command. During the redisplay process the presence of input is repeatedly checked for, and if detected causes the redisplay in progress to be aborted.

\textbf{redisplay-all} \hfill [Function]

Causes all editor windows to be completely redisplayed.
Chapter 12

Logical Characters

12.1. What a Logical Character is

Some primitives such as prompt-for-key (page 43), and commands such as EMACS query replace, read characters directly from they keyboard instead of using the command interpreter. In order to encourage consistency between these commands and make them portable and easy to customize, there is a mechanism for defining logical characters.

A logical character is a keyword which stands for some set of characters which are globally used to mean a certain thing, for example, the :help logical character stands for whatever set of characters is used to ask for help in a given implementation. It is important to note that this mapping is not a one-to-one mapping, but rather a many-to-many mapping in that a given logical character may have several corresponding real characters, and each of those characters may have several logical characters.

12.2. Logical Character Functions

*logical-character-names* [Variable]

This variable holds a string-table of all the logical characters string-names, with the values of each entry being the actual logical-character keyword.

define-logical-character string-name documentation [Function]

Takes string-name and converts it into a keyword by replacing spaces with hyphens, as with defattribute (page 32), and then defines the keyword to be a logical character having the given documentation.

logical-character-characters keyword [Function]

Returns the list of characters that are equivalent to the logical character keyword.

logical-character-name keyword [Function]

logical-character-documentation keyword [Function]

Return the string name and documentation given to define-logical-character when the logical character keyword was defined.

logical-char= character keyword [Function]

Returns true if the specified character has keyword as a corresponding logical character. The value that is returned for any character/keyword pair may be set by using setf; this is how a real character and a logical character are associated. It is a error for keyword not to be a defined logical character. character
is case-folded, thus comparisons are case insensitive, but bits and font are significant.

12.3. Standard Logical Characters

There a number of standard logical characters defined, some of which are used by functions documented in this manual, and others defined simply so that commands can use them. If a command wants to read a single character command that fits one of these descriptions then the character read should be compared to the corresponding logical character instead of wiring the actual character into the code. In many cases the command-case (page 42) macro can be used. This makes using logical characters easy, and takes care of prompting and displaying help messages.

:yes Indicates that that some action, such as doing a replacement should be taken.
:no Analogous to :yes, but it indicates that the action should not be taken.
:do-all Indicates that the action under consideration should be repeated as many times as possible.
:exit Tells the command to terminate in a normal fashion.
:help Instructs the command to display some help information.
:confirm Confirms any input, or if none, indicates that the default should be taken.
:quote Indicates that the following character is not to be treated as a command, regardless of what it is, but rather simply stands for itself.
:recursive-edit Indicates that the command should enter a recursive edit in the current context.

Define a new logical character whenever:
1. The character concerned represents a general class of actions, and thus might want to be known about by several commands.
2. The exact character chosen to invoke the action concerned is likely to be a matter of violent dispute, and thus should be easy to change.
3. The character concerned is not standard-char-p, and thus cannot be specified in a implementation independent fashion.
Chapter 13

The Echo Area

Hemlock provides a number of facilities for displaying information to and prompting the user. Most of these work through a small window displayed at the bottom of the screen. This is called the echo area.

*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. It has no modeline.

It is considered in poor taste to perform text operations on the echo area buffer to display messages -- the message function should be used instead.

Echo Area Height (initial value 3)  [Hemlock Variable]
This variable determines the initial height in lines of the echo area window.

13.1. Echo Area Clearing

clear-echo-area  [Function]
Clears the echo area.

A command must use the message function or set buffer-modified (page 10) for the Echo Area buffer to nil to leave text in the echo area after it completes.

13.2. Echo Area Functions

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.

message is usually the best way to display in the echo area since it goes to some trouble to assure that message is displayed so that it can be seen.
*echo-area-stream*  
This is a buffered Hemlock output stream 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.

13.3. 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 do some arbitrary action to help the user, and the return nil.

prompts &key :prompt :help :must-exist :default  
[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.

command-case ((key value)*) (((tag) tag) help (form)*)  
[Macro]

This macro is analogous to the COMMON LISP case macro. It is intended to be used by commands such as Query Replace which read single-character commands and dispatch from them. Since the description of this is rather complex, here is an 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))))))

Normally command-case prompts for a character, and then evaluates the first option in the body whose
tag is equivalent to the character read. Each tag may be either a logical character (page 39) or a standard character (one that satisfies the COMMON LISP standard-char-p predicate). If the tag is logical-character keyword, then it is compared to the character read with logical-char=. If the tag is a character then is case-folded and compared with char=.

The keyword arguments are used to specify how the prompting is done. The following values for a key are defined:

:help This string is displayed by the default :help option before each possibility is described.

:prompt This is the prompt used when reading the character.

:change-window If this is true (the default), then the echo area window is made the current window while the character is read. Sometimes it is desirable not to change the window since the user may want to answer the question on the basis of where the point is in the current buffer.

:bind The argument to this keyword is a variable which is to be bound to the character read.

:character If this is specified, then no character is read initially, and processing proceeds as though the character of the corresponding value had been read.

There are default options for two logical characters: :help and :abort. If a help character is read, then a help message is displayed. The message is created out of the string given to the :help key and the help strings specified for each option. After the help message is displayed the prompting is repeated. If an abort character is read then an editor error is signalled. Either of these actions may be overridden by explicitly specifying some option that subsumes these.

Instead of specifying a tag or tag list, t may be used -- this becomes the default option, and is evaluated only if no other option, including the default ones can be. This option has no help string, and is not mentioned in any help message. The default default option beeps and then does a reprompt.

Within the body of command-case, the reprompt macro is defined. Use of this macro causes the prompting and option selection process to be immediately restarted.

prompt-for-character &key :prompt :change-window [Function]
Prompts for a character and does not wait for confirmation before returning. command-case (page 42) is more useful for most purposes. When appropriate use logical characters (page 39).

prompt-for-key &key :prompt :help :must-exist :default :default-string
[Function]
Prompts for key, a vector of characters, suitable for being passed to any of the functions that manipulate key bindings (page 24). 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.

[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 not nil. prompt-for-file returns a COMMON LISP pathname.

If the file exists as entered then it is returned, otherwise it is merged with default as by merge-pathnames.
prompt-for-integer &key :prompt :help :must-exist :default [Function]
  :default-string
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 [Function]
  :default :default-string
Prompts for a keyword with completion using the string tables in the list string-tables. If must-exist is not nil then the result must be an unambiguous prefix of a string in one of the string-tables, and the complete string is returned even if only a prefix of the full string was typed. In addition, the value of the corresponding entry in the string table is returned as the second value.

If must-exist is nil then the string is returned exactly as entered. The difference between prompt-for-keyword with must-exist nil, and prompt-for-string, is that completion may be done using the Complete Parse and Complete Field commands.

prompt-for-expression &key :prompt :help :must-exist :default [Function]
  :default-string
Reads a Lisp expression. If must-exist is nil and a read error occurs then the string typed is returned.

Prompts for a string; this cannot fail.

prompt-for-variable &key :prompt :help :must-exist :default [Function]
  :default-string
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 [Function]
  :default-string
Prompts for "y" or "n" (or "Y" or "N" naturally), and returns t or nil without waiting for confirmation. When a confirming key is typed, return the default if there is one. If must-exist is nil then return whatever character was first typed if it was not "y" or "n". This is analogous to the COMMON LISP function y-or-n-p.

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

13.4. Control of Parsing Behavior

The behavior of the parsing routines is parameterized by a variable and a character attribute.

The character attribute Parse Field Separator, is a boolean attribute, a value of one indicating that that character is considered to be a field separator by the Complete Field command.
Beep On Ambiguity

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

13.5. 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*

This function, which is called by Confirm Parse (page 46), does most of the work of parsing something. The function which is bound to this variable is passed one argument, which is the string that was in *parse-input-region* when the Confirm Parse command was invoked. 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*

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

*parse-value-must-exist*

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*

The default value for this parse. If the *parse-input-region* is empty when Confirm Parse is invoked, then the string representation of this, *parse-default-string* is passed to the parse verification function.

*parse-default-string*

The string used as the printed representation of the default for the object being prompted for, e.g. when prompting for a buffer, this variable will be bound to the buffer name.

*parse-type*

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*

The prompt being used for the current parse.

*parse-help*

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

*parse-starting-mark*

This variable holds a mark in the *echo-area-buffer* (page 41) 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.

13.6. Standard Echo Area Commands

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]
Attempt to complete the current region as a keyword in *string-tables*. Give an editor-error if it 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 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
Entry Hook

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
Exit Hook

exit-hemlock leaves Hemlock and return to Lisp; value is the value to return, which defaults to t. The hook Exit Hook (page 47) is invoked before this is done.

pause-hemlock

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. Miscellaneous

beep

Causes some implementation-dependent action meant to attract attention.

14.3. 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. Redisplay is also normally done as a side-effect of keyboard input. If someone tries to read from the keyboard and there is no pending input, then redisplay is invoked.
*editor-input*  [Variable]
*real-editor-input*  [Variable]

*editor-input* is an input stream which reads characters from the keyboard immediately and without echoing.

If the eof-errorp argument to the reading function is nil then input is quoted as far as possible to enable the reading of interrupt characters and similar things.

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

editor-sleep  [Function]
Return either after time seconds have elapsed or when input is available on *editor-input*.

*character-history*  [Variable]
This is a Hemlock ring buffer (see page 52) that holds the last 60 characters read from the keyboard.

*last-character-typed*  [Variable]
This variable should be used by commands that want to know the character that invoked them. If no character has yet been typed, then the value is nil. This variable usually holds the last character read from the keyboard, but it is also maintained within keyboard macros.

*input-transcript*  [Variable]
If this is non-nil then it should be an adjustable vector with a fill-pointer. When it is non-nil all input read is also pushed onto this vector.

text-character  [Function]
When given a character as returned by reading from *editor-input*, this returns a character suitable for inserting in text, or nil if character doesn't have a text representation.

Exactly what this does is implementation dependent, but on ASCII implementations which support bits this might turn characters with the control bit on into the corresponding ASCII control character.

14.4. Hemlock Streams

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

make-hemlock-output-stream mark optional buffered  [Function]
All output directed to this stream is inserted at the permanent mark mark. buffered controls whether the stream is buffered or not. buffered may be one of 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
make-hemlock-region-stream region

Returns a stream from which the text in the region can be read.

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

While evaluating forms, binds var to a stream which returns input from region.

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

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

with-random-typeout (var n) {declaration}* {form}*

Bind var to a stream which, when output to, displays the output on the screen in some aesthetic fashion. n is an estimate of the number of lines that the output will take to display. Typically what this will do is make a window n lines high on the screen, display the output in it in more-mode, and then pause at then end until a character is typed to indicate that the input has been read. This is useful for displaying information of temporary interest such as buffer lists.

14.5. Interface to the Error System

editor-error &rest args

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. Normally editor-error is called with no arguments, in which case it will beep and abort the command in progress. If arguments are supplied then they are interpreted as format arguments for an error message to be displayed. editor-error never returns.

catch-editor-error {form}*

If an editor-error is signalled within the body of this macro, then then the execution of the forms is terminated and nil is returned, but no other action is taken. If no editor-error occurs then the value of the last form is returned.

handle-lisp-errors {form}*

Within the body of this macro any Lisp errors that occur are handled in some fashion more graceful than simple 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.

14.6. File Reading and Writing

COMMON LISP pathnames are used by the file primitives.

read-file pathname mark

Inserts the file named by pathname at mark.

write-file region pathname

Writers the contents of the region to the file named by pathname.

For probing, checking write dates, and so forth, all of the COMMON LISP file functions are available.
Chapter 15
Utilities

In this chapter, a number of utilities for manipulating some types of objects Hemlock uses to record information are given. 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.

15.1. String-table Functions

String-tables are similar to COMMON LISP hashtables in that they associate a value with an object. There are however, several 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 kind of string may be added to a string table, but the string table functions always return simple-strings.

**Function**

**make-string-table**

Make an empty string table.

**Function**

**delete-string string table**

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

**Function**

**getstring string table**

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. If set with setf a new entry is made if necessary and the old value is replaced with the new one.

**Function**

**complete-string string tables**

Returns multiple values, first the longest common prefix of all the strings in the list of tables which string is a prefix of, and if there is only one such string then the value of the corresponding entry and t are returned as the second and third values, otherwise both of these values are nil. If there is no string which string is a prefix of then all three values are nil.

**Function**

**find-ambiguous string table**

**Function**

**find-containing string table**

find-ambiguous returns a list in alphabetical order of all the strings in table which have string as a prefix. find-containing is identical except that it returns all strings which have string as a sub-string.
do-strings (string-var value-var table [result]) [declaration]* [tag I statement]* [Macro]
Iterate over the strings in table in alphabetical order. On each iteration string-var is bound to the string
for the entry and value-var is bound to the value of the entry.

15.2. Manipulating Ring Buffers

There are various purposes in an editor for which a ring-buffer can be used, so in Hemlock a general purpose ring
buffer type is provided. It can be used for such purposes as maintaining a kill-ring or a command history.

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.

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.
Index
Index

Abort Recursive Edit Hook  Hemlock variable  27
abort-recursive-edit function  27
add-hook macro  21
After Set Buffer Hook  Hemlock variable  9
Altering text  15
ask-user keyword  for make-window  36
beep function  47
Beep On Ambiguity  Hemlock variable  45
bind-key function  25
bit-prefix keys  45
blank-after-p function  13
blank-before-p function  13
blank-line-p function  13
:buffer keyword  for defhvar  19
Buffer Major Mode Hook  Hemlock variable  30
Buffer Minor Mode Hook  Hemlock variable  30
Buffer Modified Hook  Hemlock variable  10
Buffer Name Hook  Hemlock variable  10
Buffer Pathname Hook  Hemlock variable  10
buffer-end function  5
*buffer-list* variable  9,11
buffer-major-mode function  30
buffer-minor-mode function  30
buffer-modes function  30
buffer-modified function  10,41
buffer-name function  10
*buffer-names* variable  9,11
buffer-pathname function  10
buffer-point function  10
buffer-region function  10
buffer-start function  5
buffer-variables function  11
buffer-windows function  11
buffer-writable function  10
buffer-write-date function  10
Buffers  9
catch-editor-error macro  49
center-window function  37
:change-window keyword  for prompt-for-character  43
Character Attribute Hook  Hemlock variable  32
Character attributes  31
character-attribute function  32
character-attribute-char-code-limit constant  31
character-attribute-documentation function  32
character-attribute-hooks function  33
character-attribute-name function  32
*character-attribute-names* variable  31
character-attribute-p function  32
*character-history* variable  48
character-offset function  5
:clear-up function keyword  for defmode  30
clear-echo-area function  41
clear-string function  51
Command interpreter  24
command-bindings function  25
command-case macro  40,42,43
command-char-bits-limit constant  24
command-char-code-limit constant  24
command-documentation function  23
command-function function  23
command-name function  23
*command-names* variable  23,23
Commands  23
Complete Field Command  46
Complete Keyword Command  46
complete-string function  51
Confirm Parse Command  45,46
copy-mark function  4
copy-region function  6
count-characters function  7
count-lines function  6
Counting lines and characters  6
Current buffer  9
Current environment  17
Current window  35
current-buffer function  9,17,24
current-point function  9
current-window function  9,35
Cursor positions  37
cursors-to-mark function  37
defattribute function  32,39
:default keyword  for prompt-for-buffer  42
for prompt-for-expression  44
for prompt-for-file  43
for prompt-for-integer  44
for prompt-for-key  43
for prompt-for-keyword  44
for prompt-for-string  44
for prompt-for-variable  44
for prompt-for-y-or-n  44
for prompt-for-yes-or-no  44
Default Modeline Function  Hemlock variable  36
Default Modeline String  Hemlock variable  36
Default Modes  Hemlock variable  10,29
:default-string keyword  for prompt-for-buffer  42
for prompt-for-expression  44
for prompt-for-file  43
for prompt-for-integer  44
for prompt-for-key  43
for prompt-for-string  44
for prompt-for-variable  44
for prompt-for-y-or-n  44
for prompt-for-yes-or-no  44
defcommand macro  23
defhvar function  19
define-logical-character function  39
defmode function  26,30
Delete Buffer Hook  Hemlock variable  11
Delete Variable Hook  Hemlock variable  20
Delete Window Hook  Hemlock variable  36
delete-and-save-region function  15
delete-buffer function  11
delete-characters function  15
delete-key-binding function  25
delete-mark function  5
delete-region function  15
delete-string function  51
delete-variable function  20
delete-window function  36
Deleting  15
displayed-p function  37
do-strings macro  52

55
INDEX

move-to-column function 38
move-to-position function 5, 10, 38
Moving marks 5
:must-exist keyword
  for prompt-for-buffer 42
  for prompt-for-expression 44
  for prompt-for-file 43
  for prompt-for-integer 44
  for prompt-for-key 43
  for prompt-for-keyword 44
  for prompt-for-variable 44
  for prompt-for-y-or-n 44
  for prompt-for-yes-or-no 44

new-search-pattern function 16
next-character function 4
next-window function 37

nineteen-region function 15

*parse-default* variable 45
*parse-default-string* variable 45
*parse-help* variable 45
*parse-input-region* variable 46
*parse-prompt* variable 45
*parse-string-tables* variable 45
*parse-type* variable 45
*parse-value-must-exist* variable 45
*parse-verification-function* variable 45
pause-hemlock function 47
Permanent marks 4
:precedence keyword
  for defmode 30
Prefix arguments 26
prefix-argument function 27
previous-character function 4
previous-window function 37

:prompt keyword
  for prompt-for-buffer 42
  for prompt-for-character 43
  for prompt-for-expression 44
  for prompt-for-file 43
  for prompt-for-integer 44
  for prompt-for-key 43
  for prompt-for-keyword 44
  for prompt-for-string 44
  for prompt-for-variable 44
  for prompt-for-y-or-n 44
  for prompt-for-yes-or-no 44
prompt-for-buffer function 42
prompt-for-character function 43
prompt-for-expression function 44
prompt-for-file function 43
prompt-for-integer function 44
prompt-for-key function 39, 43
prompt-for-keyword function 44
prompt-for-string function 44
prompt-for-variable function 44
prompt-for-y-or-n function 44
prompt-for-yes-or-no function 44
Prompting functions 42

read-file function 49
*real-editor-input* variable 48
Recursive edit 27
recursive-edit function 24, 27
redisplay function 38
redisplay-all function 38
region function 6
region-bounds function 6

region-end function 6
region-start function 6
region-to-string function 6
Regions 6
remove-hook macro 21, 33
replace-pattern function 16
Replacing 16
reverse-find-attribute function 32
Ring Buffers 52
ring-length function 52
ring-pop function 52
ring-push function 52
ring-read function 52
rotate-ring function 52

same-line-p function 13
scroll-window function 37
search-char-code-limit constant 16
Searching 16
Set Buffer Hook Hemlock variable 9
Set Window Hook Hemlock variable 35
set-region-bounds function 6
:setup-function keyword
  for defmode 30
setv macro 20
Shadow Attribute Hook Hemlock variable 32
shadow-attribute function 32
show-mark function 38
start-line-p function 13
String-tables 51
string-to-region function 6
string-to-variable function 20
Syntax tables 31

Temporary marks 4
text-character function 48
translating keys 25
Transparent key bindings 26
:transparent-p keyword
  for defmode 30
Unshadow Attribute Hook Hemlock variable 32
unshadow-attribute function 32
update-window-modeline function 35
use-buffer macro 27
Utilities 51

:value keyword
  for defhvar 19
value macro 20
variable-documentation function 20
variable-hooks function 20
variable-name function 20
variable-value function 20

Window Buffer Hook Hemlock variable 36
window-buffer function 36
window-display-end function 36
window-display-start function 36
window-height function 37
*window-list* variable 36
window-modeline-function function 35
window-modeline-string function 35
window-point function 36
window-width function 37
Windows 35

with-input-from-region macro 49
with-mark macro 5
with-output-to-mark macro 49
with-random-typeout macro 49
write-file function 49