1 Introduction 1
   Using the editor within LispWorks 2

2 General Concepts 5
   Window layout 5
   Buffer locations 7
   Modes 8
   Text handling concepts 8
   Executing commands 9
   Basic editing commands 11

3 Command Reference 15
   Aborting commands and processes 16
   Executing commands 18
   Help 18
   Prefix arguments 24
   File handling 25
   Movement 37
   Marks and regions 44
   Locations 47
   Deleting and killing text 48
   Inserting text 53
   Delete Selection 56
   Undoing 56
Contents

Key bindings 160
Replacing the current selection 161
Emulation in Applications 161

6 Advanced Features 163
Customizing default key bindings 164
Customizing Lisp indentation 166
Programming the editor 166
Editor source code 195

Glossary 197

Index 207
1

Introduction

The LispWorks editor is built in the spirit of Emacs. As a matter of policy, the key bindings and the behavior of the LispWorks editor are designed to be as close as possible to the standard key bindings and behavior of GNU Emacs.

For users more familiar with KDE/Gnome keys, an alternate keys and behavior model is provided. This manual however, generally documents the Emacs model.

The LispWorks editor has the following features:

- It is a *screen* editor. This means that text is displayed by the screenful, with a screen normally displaying the text which is currently being edited.
- It is a *real-time* editor. This means that modifications made to text are shown immediately, and any commands issued are executed likewise.
- An *on-line help* facility is provided, which allows the user quick and easy access to command and variable definitions. Various levels of help are provided, depending on the type of information the user currently possesses.
- It is *customizable*. The editor can be customized both for the duration of an editing session, and on a more permanent basis.
- A range of commands are provided to facilitate the editing of Lisp programs.
• The editor is itself written in Lisp.

1.1 Using the editor within LispWorks

The LispWorks editor is fully integrated into the LispWorks programming environment. If you don’t currently have an Editor (check the Windows menu), start one by choosing Tools > Editor from the podium or clicking on in the podium toolbar.

There are a number of editor operations which are only available in Listener windows (for example, operations using the command history). These operations are covered in the LispWorks IDE User Guide.

1.1.1 About this manual

The Editor User Guide is divided into chapters, as follows:

Chapter 2, “General Concepts”, provides a brief overview of terms and concepts which the user should be familiar with before progressing to the rest of the manual. The section ‘Basic editing commands’ provides a brief description of commands necessary to edit a file from start to finish. If you are already familiar with Emacs, you should be familiar with most of the information contained in this chapter.

Chapter 3, “Command Reference”, contains full details of most of the editor commands. Details of editor variables are also provided where necessary. Not included in this chapter are commands used to facilitate the editing of Lisp programs.

Chapter 4, “Editing Lisp Programs”, contains full details of editor commands (and variables where necessary) to allow for easier editing of Lisp programs.

Chapter 5, “Emulation”, describes use of KDE/Gnome style key bindings rather than Emacs style.

Chapter 6, “Advanced Features”, provides information on customizing and programming the editor. The features described in this chapter allow permanent changes to be made to the editor.

A “Glossary” is also included to provide a quick and easy reference to editor terms and concepts.
Each editor command, variable and function is fully described once in a relevant section (for example, the command **Save File** is described in “File handling” on page 12). It is often worthwhile reading the introductory text at the start of the section, as some useful information is often provided there. The descriptions all follow the same layout convention which should be self-explanatory.

Command description layouts include the name of the command, the default Emacs binding, details of optional arguments required by the associated defining function (if any) and the mode in which the command can be run (if not global).
1 Introduction
There are a number of terms used throughout this manual which the user should be familiar with. Definitions of these terms are provided in this chapter, along with a section containing just enough information to be able to edit a document from start to finish.

This chapter is not designed to provide precise details of commands. For these see the relevant sections in the following chapters.

2.1 Window layout

2.1.1 Windows and panes

When the editor is called up an editor window is created and displayed (for those already familiar with Emacs running on a tty terminal, note that in this context a window is an object used by the window manager to display data, and not a term used to describe a portion of the editor display). The largest area of the editor window is taken up by an editor pane. Each window contains a single pane and therefore the term window is used throughout this manual as being synonymous with pane, unless more clarification is required.

Initially only one editor window is displayed. The corresponding editor pane is either blank (ready for text to be entered) or contains text from a file to be
2 General Concepts

edited. The editor window displays text using the font associated with the editor pane.

2.1.2 Files and buffers

It is not technically correct to say that a window displays the contents of a file, rather that each window displays the contents of a buffer. A buffer is an object that contains data from the point of view of the editor, whereas a file contains data from the point of view of the operating system. A buffer is a temporary storage area used by the editor to hold the contents of a file while the process of editing is taking place. When editing has finished the contents of the buffer can then be written to the appropriate file. When the user exits from the editor, no information concerning buffers or windows is saved.

A buffer is often displayed in its own window, although it is also possible for many buffers to be associated with a single window, and for a single buffer to be displayed in more than one window.

In most cases, there is one buffer for each file that is accessed, but sometimes there is more than one buffer for a single file. There are also some buffers (such as the Echo Area, which is used to communicate with the user) that are not necessarily associated with any file.

2.1.3 The mode line

At the bottom of each editor window is a mode line that provides information concerning the buffer which that window is displaying. The contents of the mode line are as follows:

- "----" or "*-*" or "%-%": the first indicates that the buffer is unchanged since it was last saved; the second that it has been changed; and the third that it is read only.
- the name of the buffer (the name of a buffer originating from a file is usually the same as the name of that file).
- the package of the current buffer written within braces.
- a major mode (such as Fundamental or Lisp). An buffer always operates in a single major mode.
2.2 Buffer locations

2.2.1 Points

A point is a location in a buffer where editor commands take effect. The current point is generally the location between the character indicated by the cursor and the previous character (that is, it actually lies between two characters). Many types of commands (that is, moving, inserting, deleting) operate with respect to the current point, and indeed move that point.

Each buffer has a current point associated with it. A buffer that is not being displayed remembers where its current point is and returns the user to that point when the buffer is redisplayed.

If the same buffer is being displayed in more than one window, there is a point associated with the buffer for each window. These points are independent of each other.

2.2.2 Marks

The location of a point can be saved for later reference by setting a mark. Marks may either be set explicitly or as side effects of commands. More than one mark may be associated with a single buffer and saved in what is known as a mark ring. As for points, the positions of marks in a buffer are remembered even if that buffer is not currently being displayed.
2.2.3 Regions

A region is the area of text between the mark and the current point. Many editor commands affect only a specified region.

2.3 Modes

Each buffer can be in two kinds of mode: a major mode, such as Lisp mode, or Fundamental mode (which is the ordinary text processing mode); and a minor mode, such as Abbrev mode or Auto-Fill mode. A buffer always has precisely one major mode associated with it, but minor modes are optional. Any number of minor modes can be associated with a buffer.

The major modes govern how certain commands behave. For example, the concept of indentation is radically different in Lisp mode and in Fundamental mode. When a file is loaded into a new buffer, the default mode of that buffer is determined by the file name. For example, a buffer into which a file name that has a .lisp suffix is loaded defaults to Lisp mode.

The minor modes determine whether or not certain actions take place. For example, when Auto-Fill mode is on lines are automatically broken at the right hand margin, as the text is being typed, when the line length exceeds a predefined limit. Normally the newline has to be entered manually at the end of each line.

2.4 Text handling concepts

2.4.1 Words

A word is defined as a continuous string of alphanumeric characters. These are the letters A-Z, a-z, numbers 0-9, and the Latin-1 alphanumeric characters). In most modes, any character which is not alphanumeric is treated as a word delimiter.

2.4.2 Sentences

A sentence begins wherever a paragraph or previous sentence ends. The end of a sentence is defined as consisting of a sentence terminating character followed by two spaces or a newline. Two spaces are required to prevent abbreviations...
2.5 Executing commands

(such as Mr.) from being taken as the end of a sentence. Such abbreviations at the end of a line are taken as the end of a sentence. There may also be any number of closing delimiter characters between the sentence terminating character and the spaces or newline.

Sentence terminating characters include: . ? !

Closing delimiter characters include: ) ] > / | " '

2.4.3 Paragraphs

A paragraph is defined as the text within two paragraph delimiters. A blank line constitutes a paragraph delimiter. The following characters at the beginning of a line are also paragraph delimiters:

Space Tab @ - ’ )

2.5 Executing commands

2.5.1 Keys — Ctrl and Meta

Editor commands are initiated by one or more key sequences. A single key sequence usually involves holding down one of two specially defined modifier keys, while at the same time pressing another key which is usually a character key.

The two modifier keys referred to are the Control (Ctrl) key and the Meta key.

When using Emacs emulation on a keyboard without a Meta key, the Escape (Esc) key can be used instead. Note that Esc must be typed before pressing the required character key, and not held down.

When using KDE/Gnome editor emulation Esc is the cancel gesture, so LispWorks provides an alternate gesture to access editor commands: Ctrl+M. For example, to invoke the command Find Source for Dspec, type

```
Ctrl+M X Find Source for Dspec
```

and press Return.

To continue the search, type Ctrl+M ,.
An example of a single key sequence command is Ctrl+A which moves the current point to the start of the line. This command is issued by holding down the Control key while at the same time pressing A.

Some key sequences may require more than one key sequence. For example, the key sequence to save the current buffer to a file is Ctrl+X Ctrl+S. Another multi-key sequence is Ctrl+X S which saves all buffers to their relevant files. Note that in this case you do not press the Control key while pressing S.

A few commands require both the Ctrl and Meta key to be held down while pressing the character key. Meta+Ctrl+L, used to select the previous buffer displayed, is one such command. If the Esc key is being used in place of the Meta key, then this key should be pressed before the Ctrl+L part of the key sequence.

There is a key sequence for which you cannot use Esc instead of Meta, because it is not actually implement as an editor command (it works in other windows too). This is the default break gesture Meta+Ctrl+C described in “Aborting commands and processes” on page 16. As there are so many different types of keyboard, if it is not possible to assert which is the Meta key on your keyboard, it may be marked with a special character, such as a diamond, or it may be one of the function keys — try F11. From this point on we refer exclusively to the Meta key in this manual.

### 2.5.2 Two ways to execute commands

The key sequences used to execute commands, as described in the previous section, are only one way to execute an editor command. As a general rule, editor commands that are used frequently should involve as few key strokes as possible to allow for fast editing. The key sequences described above are quick and easy shortcuts for invoking commands.

Most editor commands can also be invoked explicitly by using their full names. For example, in the previous section we met the keystroke Ctrl+A which moves the current point to the beginning of the line. This keystroke is called a key binding and is a shortcut for executing the command Beginning of Line. To execute this command by name you must type Meta+X followed by the full command name (Meta+X itself is only a key binding for the command Extended Command).

Even though there may seem like a lot of typing to issue the extended version of a command, it is not generally necessary to type in the whole of a command to
be executed. The Tab key can be used to complete a partially typed in extended command. The editor extends the command name as far as possible when Tab is used, and if the user is not sure of the rest of the command name, then pressing Tab again provides a list of possible completions. The command can then be selected from this list.

The most commonly used editor commands have a default binding associated with them.

2.5.3 Prefix arguments

Editor commands can be supplied with an integer argument which sometimes alters the effect of that command. In most cases it means the at the command is repeated that many times. This argument is known as a prefix argument as it is supplied before the command to which it is to be applied. Prefix arguments sometimes have no effect on a command.

2.6 Basic editing commands

This section contains just enough information to allow you to load a file into the editor, edit that file as required, and then save that file. It is designed to give you enough information to get by and no more.

Only the default bindings are provided. The commands introduced are grouped together as they are in the more detailed command references and under the same headings (except for “Killing and Yanking” on page 13). For further information on the commands described below and other related commands, see the relevant sections in Chapter 3, Command Reference.

2.6.1 Aborting commands and processes

See “Aborting commands and processes” on page 16

Ctrl+G  Abort the current command which may either be running or just partially typed in. Use Esc in KDE/Gnome editor emulation.
2.6.2 File handling

See “File handling” on page 25.

- \texttt{Ctrl+X Ctrl+F file}
  - Load file into a buffer ready for editing. If the name of a non-existent file is given, then an empty buffer is created in to which text can be inserted. Only when a save is done will the file be created.

- \texttt{Ctrl+X Ctrl+S}
  - Save the contents of the current buffer to the associated file. If there is no associated file, one is created with the same name as the buffer.

2.6.3 Inserting text

See “Inserting text” on page 53 for details of various commands which insert text.

Text which is typed in at the keyboard is automatically inserted to the left of the cursor.

To insert a newline press \texttt{Return}.

2.6.4 Movement

See “Movement” on page 37.

- \texttt{Ctrl+F}
  - Move the cursor forward one character.

- \texttt{Ctrl+B}
  - Move the cursor backward one character.

- \texttt{Ctrl+N}
  - Move the cursor down one line.

- \texttt{Ctrl+P}
  - Move the cursor up one line.

The above commands can also be executed using the arrow keys.

- \texttt{Ctrl+A}
  - Move the cursor to the beginning of the line.

- \texttt{Ctrl+E}
  - Move the cursor to the end of the line.

- \texttt{Ctrl+V}
  - Scroll one screen forward.

- \texttt{Meta+V}
  - Scroll one screen backward.
2.6 Basic editing commands

Meta+Shift+<  Move to the beginning of the buffer.
Meta+Shift+>  Move to the end of the buffer.

2.6.5 Deleting and killing text
See “Deleting and killing text” on page 48.

Delete  Delete the character to the left of the cursor.
Ctrl+D  Delete the current character.
Ctrl+K  Kill text from the cursor to the end of the line. To delete a whole line (that is, text and newline), type Ctrl+K twice at the start of the line.

2.6.6 Undoing
See “Undoing” on page 56.

Ctrl+Shift+ _  Undo the previous command. If Ctrl+Shift+ _ is typed repeatedly, previously executed commands are undone in a “last executed, first undone” order.

2.6.7 Killing and Yanking
The commands given below are used to copy areas of text and insert them at some other point in the buffer. Note that there is no corresponding “Cut and paste” section in the command references, so direct cross references have been included with each command.

When cutting and pasting, the first thing to do is to copy the region of text to be moved. This is done by taking the cursor to the beginning of the piece of text to be copied and pressing Ctrl+Space to set a mark, and then taking the cursor to the end of the text and pressing Ctrl+W. This kills the region between the current point and the mark but keeps a copy of the killed text. This copy can then be inserted anywhere in the buffer by putting the cursor at the required position and then pressing Ctrl+Y to insert the copied text.

If the original text is to be copied but not killed, use the command Meta+W instead of Ctrl+W. This copies the text ready for insertion, but does not delete it.
2 General Concepts

**Ctrl+Space**  Set a mark for a region. See “Marks and regions” on page 44.

**Ctrl+W**  Kill the region between the mark and current point, and save a copy of that region. See “Deleting and killing text” on page 48.

**Meta+W**  Copy the region between the mark and the current point. See “Deleting and killing text” on page 48.

**Ctrl+Y**  Insert (yank) a copied region before the current point. See “Inserting text” on page 53.

### 2.6.8 Help

See “Help” on page 18.

**Ctrl+H A string**  List all commands whose name contains `string`.

**Ctrl+H D command**  Describe `command`, where `command` is the full command name.

**Ctrl+H K key**  Describe the command bound to `key`. 
This chapter contains full details of most of the editor commands. Details of related editor variables have also been included alongside commands, where appropriate. Not included in this chapter, are commands used to facilitate the editing of Lisp programs. See Chapter 4, Editing Lisp Programs.

Commands are grouped according to functionality as follows:

- "Aborting commands and processes"
- "Executing commands"
- "Help"
- "Prefix arguments"
- "File handling"
- "Movement"
- "Marks and regions"
- "Deleting and killing text"
- "Inserting text"
- "Undoing"
- "Case conversion"
3.1 Aborting commands and processes

**Key Sequence**

Ctrl+G

Aborts the current command. Ctrl+G (or Esc in KDE/Gnome editor emulation) can either be used to abandon a command which has been partially typed in, or to abort the command which is currently running.
3.1 Aborting commands and processes

Note that, unlike most of the keys described in this manual, this cannot be changed via `editor:bind-key`. Instead, use `editor:set-interrupt-keys` if you wish to change this.

**Key Sequence**

Meta+Ctrl+C

Chooses a process that is useful to break, and breaks it.

Note that you cannot use `Escape` in place of Meta. As there are many different types of keyboard, if it is not possible to assert which is the Meta key on your keyboard, it may be marked with a special character, such as a diamond, or it may be one of the function keys — try F11.

Meta+Ctrl+C applies to both GTK+ and Motif. If your keyboard has the Break key, then you can also use this alternate break gesture. The key sequence can be configured using `capi:set-interactive-break-gestures`.

The process to break is chosen as follows:

1. If the break gesture is sent to any CAPI interface that is waiting for events, it does "Interface break", as described below.

2. Otherwise it checks for a busy processes that is essential for LispWorks to work correctly, or that interacts with the user (normally that means that some CAPI interface uses it), or that is flagged as wanting interrupts (currently that means a REPL). If it finds such a busy process, it breaks it.

3. Otherwise, if the LispWorks IDE is running, activate or start the Process Browser. Note that the Process Browser tool, documented in the *LispWorks IDE User Guide* can be used to break any other process.

4. Otherwise, if there is a busy process break it.

5. Otherwise, just break the current process.

"Interface break" depends on the interface. For an interface that has another process, notably the Listener with its REPL, it breaks that other process. For most interfaces, in the LispWorks IDE it starts the Process Browser, otherwise just it breaks the interface's process.
3.2 Executing commands

Some commands (usually those used most frequently) are bound to key combinations or key sequences, which means that fewer keystrokes are necessary to execute these commands. Other commands must be invoked explicitly, using Extended Command.

It is also possible to execute shell commands from within the editor. See “Running shell commands from the editor” on page 110.

Extended Command

Key sequence: Meta+X

Allows the user to type in a command explicitly. Any editor command can be invoked in this way, and this is the usual method of invoking a command that is not bound to any key sequence. Any prefix argument is passed to the command that is invoked.

It is not generally necessary to type in the whole of a command to be executed. Completion (using Tab) can be used after the first part of the command has been typed.

3.3 Help

The editor provides a number of on-line help facilities, covering a range of areas.

There is one main help command, accessed by Help (Ctrl+H), with many options to give you a wide range of help on editor commands, variables and functions.

There are also further help commands which provide information on Lisp symbols (see “Documentation” on page 145).
3.3 Help

3.3.1 The help command

Help Editor Command

Options: See below
Key sequence: \texttt{Ctrl+H option}

Provides on-line help. Depending on what information the user has and
the type of information required, one of the following options should be
selected after invoking the \texttt{Help} command. In most cases a \texttt{Help} command
plus option can also be invoked by an extended editor command.

A brief summary of the help options is given directly below, with more
detailed information following.

\begin{itemize}
  \item \texttt{?} Display a list of help options.
  \item \texttt{q or n} Quit help.
  \item \texttt{a string} Display a list of commands whose names contain \texttt{string}.
  \item \texttt{b} Display a list of key bindings and associated commands.
  \item \texttt{c key} Display the command to which \texttt{key} is bound.
  \item \texttt{d command} Describe the editor \texttt{command}.
  \item \texttt{Ctrl+D command} Bring up the on-line version of this manual for \texttt{command}.
  \item \texttt{g object} Invoke the appropriate describe \texttt{object} command.
  \item \texttt{k key} Describe the command to which \texttt{key} is bound.
  \item \texttt{Ctrl+K key} Bring up the on-line version of this manual for \texttt{key}.
  \item \texttt{l} describe the last 60 keys typed.
  \item \texttt{v variable} Describe \texttt{variable} and show its current value.
\end{itemize}
Ctrl+V variable  Bring up the on-line version of this manual for variable.

w command  Display the key sequence to which command is bound.

### Apropos Command

Arguments: string

Key sequence: Ctrl+H A string

Displays a list of editor commands, variables, and attributes whose names contain string, in a Help window.

Editor command, variable and attribute names tend to follow patterns which becomes apparent as you look through this manual. For example, commands which perform operations on files tend to contain the string file, that is, Find File, Save File, Print File and so forth.

Use this form of help when you know what you would like to do, but do not know a specific command to do it.

### What Command

Arguments: key

Key sequence: Ctrl+H C key

Displays the command to which key is bound. For a more detailed description of key use the command Describe Key.

Use this form of help when you know a default binding but want to know the command name.

*Note:* this command is also available via the menu command Help > Editing > Key to Command.

### Describe Command

Arguments: command

Key sequence: Ctrl+H D command
3.3 Help

Describes the editor command `command`. Full documentation of that command is printed in a Help window.

Use this form of help when you know a command name and require full details of that command.

**Document Command**

*Editor Command*

Arguments: `command`

Key sequence: `Ctrl+H Ctrl+D command`

Brings up the on-line version of this manual at the entry for `command`.

The documentation in the on-line manual differs from the editor on-line help (as produced by **Describe Command**), but provides similar information. If you are used to the layout and definitions provided in this manual then use this help command instead of `Ctrl+H D`.

**Generic Describe**

*Editor Command*

Arguments: `object`

Key sequence: `Ctrl+H G object`

Describes `object`, where `object` may take the value `command`, `key`, `attribute` or `variable`.

If `object` is `command`, `key` or `variable` then the command **Describe Command**, **Describe Key** or **Describe Editor Variable** is invoked respectively.

There is no corresponding describe command if the object is `attribute`. Attributes are things such as word delimiters, Lisp syntax and parse field separators. If you are not sure of the attributes documented remember that you can press `Tab` to display a completion list.

**Describe Key**

*Editor Command*

Arguments: `key`

Key sequence: `Ctrl+H K key`

Describes the command to which `key` is bound. Full documentation of that command is printed in a Help window.
Use this form of help when you know a default binding and require the command name plus full details of that command.

**Document Key**

*Editor Command*

Arguments: *key*

Key sequence: $\text{C}tr\text{l}+H \ Ctr\text{l}+K \ key$

Brings up the on-line version of this manual at the entry for *key*.

The documentation in the on-line manual differs slightly from the editor on-line help but usually provides you with the same amount of information. If you are used to the layout and definitions provided in this manual then use this help command instead of **Describe Key**.

**What Lossage**

*Editor Command*

Arguments: None

Key sequence: $\text{C}tr\text{l}+H \ L$

Displays the last 60 keys typed.

**Describe Editor Variable**

*Editor Command*

Arguments: *variable*

Key sequence: $\text{C}tr\text{l}+H \ V \ variable$

Describes *variable* and prints its current value in a Help window.

Use this form of help when you know a variable name and require a description of that variable and/or its current value.

**Document Variable**

*Editor Command*

Arguments: *variable*

Key sequence: $\text{C}tr\text{l}+H \ Ctr\text{l}+V \ variable$

Brings up the on-line version of this manual at the entry for *variable*.

The documentation in the on-line manual differs slightly from the editor on-line help but usually provides you with the same amount of informa-
3.3 Help

If you are used to the layout and definitions provided in this manual then use this help command instead of Describe Editor Variable.

Where Is

*Editor Command*

Arguments: *command*

Key sequence: Ctrl+H W *command*

Displays the key sequence to which *command* is bound.

Use this form of help if you know a command name and wish to find the bindings for that command. If no binding exists then a message to this effect is returned.

**Note:** this command is also available via the menu command Help > Editing > Command to Key.

Describe Bindings

*Editor Command*

Arguments: None

Key sequence: Ctrl+H B

Displays a list of key bindings and associated commands in a Help window. First the minor and major mode bindings for the current buffer are printed, then the global bindings.

3.3.2 Other help commands

Manual Entry

*Editor Command*

Arguments: *unix-command*

Key sequence: None

Displays the UNIX manual page for *unix-command*. The UNIX utility *man* is invoked and the manual page is displayed in an Editor window.

With no prefix argument, the same buffer is used each time. With a prefix argument, a new buffer is created for each manual page accessed.
3.4 Prefix arguments

Editor Commands can be supplied with an integer argument which, in many cases, indicates how many times a command is to be executed. This argument is known as a prefix argument as it is supplied before the command to which it is to be applied.

A prefix argument applied to some commands has a special meaning. Documentation to this effect is provided with the command definitions where appropriate in this manual. In most other cases the prefix argument repeats the command a certain number of times, or has no effect.

A prefix argument can be supplied to a command by first using the command Set Prefix Argument (Ctrl+U) followed by an integer. Negative prefix arguments are allowed. A prefix argument between 0 and 9 can also be supplied using Meta+digit.

Set Prefix Argument

Editor Command

Arguments: [integer]
Key sequence: Ctrl+U [integer]

Provides a prefix argument which, for many commands, indicates the command is to be invoked integer times. The required integer should be input and the command to which it applies invoked without an intervening carriage return.

If no integer is given, the prefix argument defaults to the value of prefix-argument-default.

If Set Prefix Argument is invoked more than once before a command, the prefix arguments associated with each invocation are multiplied together and the command to which the prefix arguments are to be applied is repeated this number of times. For example, if you typed in Ctrl+U Ctrl+U 2 before a command, then that command would be repeated 8 times.

prefix-argument-default

Editor Variable

Default value: 4
3.5 File handling

The default value for the prefix argument if no integer is provided for Set Prefix Argument.

None

Key sequence: Meta+<0–9>

Provides a prefix argument in a similar fashion to Set Prefix Argument, except that only integers from 0 to 9 can be used (unless the key bindings are changed).

Negative Argument

Editor Command

Arguments: None
Key sequence: None

Negates the current prefix argument. If there is currently no prefix argument then it is set to -1.

There is rarely any need for explicit use of this command. Negative prefix arguments can be entered directly with Set Prefix Argument by typing a - before the integer.

3.5 File handling

This section contains details of commands used for file handling.

The first section provides details on commands used to copy the contents of a file into a buffer for editing, while the second deals with copying the contents of buffers to files.

You may at some point have seen file names either enclosed in # characters or followed by a ~ character. These files are created by the editor as backups for the file named. The third section deals with periodic backups (producing file names enclosed in #) and the fourth with backups on file saving (producing files followed by ~).

There are many file handling commands which cannot be pigeon-holed so neatly and these are found in the section “Miscellaneous file operations” on
3 Command Reference

page 35. Commands use to print, insert, delete and rename files are covered here, along with many others.

3.5.1 Finding files

Find File

Arguments: pathname
Key sequence: None

editor:find-file-command p &optional pathname

Finds a new buffer with the same name as pathname (where pathname is the name of the file to be found, including its directory relative to the current directory), creating it if necessary, and inserts the contents of the file into the buffer. The contents of the buffer are displayed in an editor pane and may then be edited.

The file is initially read in the external format (encoding) given by the editor variable input-format-default. If the value of this is nil, cl:open chooses the external format to use. The external format is remembered for subsequent reading and writing of the buffer.

If the file is already being visited a new buffer is not created, but the buffer already containing the contents of that file is displayed instead.

If a file with the specified name does not exist, an empty buffer with that file name is created for editing purposes, but the new file is not created until the appropriate save file command is issued.

If there is no prefix argument, a new Editor window is created for the file. With any prefix argument, the file is shown in the current window.

Another version of this command is Wfind File which is usually used for finding files.

Wfind File

Arguments: pathname
Key sequence: Ctrl+X Ctrl+F pathname
3.5 File handling

editor:wfind-file-command p &optional pathname

Calls Find File with a prefix argument (that is, the new file is opened in the existing window).

Visit File  Editor Command

Arguments: pathname
Key sequence: None

editor:visit-file-command p &optional pathname buffer

Does the same as Find Alternate File, and then sets the buffer to be writable.

Find Alternate File  Editor Command

Arguments: pathname
Key sequence: Ctrl+X Ctrl+V pathname

editor:find-alternate-file-command p &optional pathname buffer

Does the same as Find File with a prefix argument, but kills the current buffer and replaces it with the newly created buffer containing the file requested. If the contents of the buffer to be killed have been modified, the user is asked if the changes are to be saved to file.

The argument buffer is the buffer in which the contents of the file are to be displayed. buffer defaults to the current buffer.

The prefix argument is ignored.

input-format-default  Editor Variable

Default value: nil

The default external format used by Find File, Wfind File and Visit File for reading files into buffers.

If the buffer already has an external format (either it has previously been read from a file, or Set External Format has been used to specify an external format) then input-format-default is ignored. If the value is nil
and the buffer does not have an external format, `cl:open` chooses the external format to use.

The value should be `nil` or an external format specification. See the *Lisp-Works User Guide and Reference Manual* for a description of these and of how `cl:open` chooses an external format.

If you have specified an input encoding via the Editor tool’s Preferences dialog, then `input-format-default` is initialized to that value on startup.

### 3.5.2 Saving files

**Save File**

*Editor Command*

Arguments: None  
Key sequence: `Ctrl+X Ctrl+S`

```
editor:save-file-command p &optional buffer
```

Saves the contents of the current buffer to the associated file. If there is no associated file, one is created with the same name as the buffer, and written in the same encoding as specified by the editor variable `output-format-default`, or as defaulted by `open` if this is `nil`.

The argument `buffer` is the buffer to be saved in its associated file. The default is the current buffer.

**Save All Files**

*Editor Command*

Arguments: None  
Key sequence: `Ctrl+X S`

Without a prefix argument, a Save Selected Buffers dialog is displayed asking whether each modified buffer is to be saved. If a buffer has no associated file it is ignored, even if it is modified. The selected buffers are saved.

With a non-nil prefix argument, no such dialog is displayed and all buffers that need saving are saved. You can also prevent the Save Selected Buffers dialog from being displayed by setting the value of the editor variable `save-all-files-confirm`. 
3.5 File handling

**save-all-files-confirm**

*Editor Variable*

- Default value: `t`

When the value is true, **Save All Files** prompts for confirmation before writing the modified buffers, when used without a prefix argument.

**Write File**

*Editor Command*

- Arguments: `pathname`
- Key sequence: `Ctrl+X Ctrl+W pathname`

```lisp
(editor:write-file-command p &optional pathname buffer)
```

Writes the contents of the current buffer to the file defined by `pathname`. If the file already exists, it is overwritten. If the file does not exist, it is created. The buffer then becomes associated with the new file.

The argument `buffer` is the name of the buffer whose contents are to be written. The default is the current buffer.

**Write Region**

*Editor Command*

- Arguments: `pathname`
- Key sequence: None

```lisp
(editor:write-region-command p &optional pathname)
```

Writes the region between the mark and the current point to the file defined by `pathname`. If the file already exists, it is overwritten. If the file does not exist, it is created.

**Append to File**

*Editor Command*

- Arguments: `pathname`
- Key sequence: None

Appends the region between the mark and the current point to the file defined by `pathname`. If the file does not exist, it is created.
3 Command Reference

**Backup File**

*Editor Command*

**Arguments:** pathname

**Key sequence:** None

Writes the contents of the current buffer to the file defined by *pathname*. If the file already exists, it is overwritten. If it does not exist, it is created.

In contrast with *Write File*, no change is made concerning the file associated with the current buffer as this command is only intended to be used to write the contents of the current buffer to a backup file.

**Save All Files and Exit**

*Editor Command*

**Arguments:** None

**Key sequence:** Ctrl+X Ctrl+C

A *Save Selected Buffers* dialog is displayed asking whether each modified buffer is to be saved. If a buffer has no associated file it is ignored, even if it is modified (this operates just like *Save All Files*). When all the required buffers have been saved LispWorks exits, prompting for confirmation first.

**add-newline-at-eof-on-writing-file**

*Editor Variable*

Default value: :ask-user

Controls whether the commands *Save File* and *Write File* add a newline at the end of the file if the last line is non-empty.

If the value of this variable is t then the commands add a newline and tell the user. If its value is nil the commands never add a newline.

If the value is t then the commands add a newline and tell the user. If the value is nil the commands never add a newline. If the value is :ask-user, the commands ask whether to add a newline.

**output-format-default**

*Editor Variable*

Default value: nil

The default external format used for writing buffers to files.
If the buffer already has an external format (either it has been read from a file, or Set External Format has been used to specify an external format) then output-format-default is ignored. If the value is nil and the buffer does not have an external format, cl:open chooses the external format to use.

The value should be nil or an external format specification. See the Lisp-Works User Guide and Reference Manual for a description of these and of how cl:open chooses an external format.

If you have specified an output encoding via the Editor tool’s Preferences dialog, then output-format-default is initialized to that value on startup.

The default value of output-format-default is nil.

Set External Format

Editor Command

Arguments: buffer
Key sequence: None

Prompts for an external format specification, providing a default which is the buffer’s current external format if set, or the value of output-format-default. Sets the buffer’s external format, so that this is used for subsequent file writing and reading.

If a non-nil prefix argument is supplied, the buffer’s external format is set to the value of output-format-default without prompting.


Find Unwritable Character

Editor Command

Arguments: None
Key sequence: None

Finds the next occurrence of a character in the current buffer that cannot be written using the buffer external format. The prefix argument is ignored.
List Unwritable Characters

*Editor Command*

Arguments: None
Key sequence: None

Lists the characters in the current buffer that cannot be written with the buffer external format. The prefix argument is ignored.

### 3.5.3 Auto-saving files

The auto-save feature allows for periodic backups of the file associated with the current buffer. These backups are only made if auto-save is switched on.

This feature is useful if the LispWorks editor is killed in some way (for example, in the case of a system crash or accidental killing of the editor process) before a file is explicitly saved. If automatic backups are being made, the state of a file when it was last auto-saved can subsequently be recovered.

By default, automatic backups are made both after a predefined number of key strokes, and also after a predefined amount of time has elapsed.

By default, auto-saved files are in the same directory as the original file, with the name of the auto-save file (or "checkpoint file") being the name of the original file enclosed within # characters.

**Toggle Auto Save**

*Editor Command*

Arguments: None
Key sequence: None

Switches auto-save on if it is currently off, and off if it is currently on.

With a positive prefix argument, auto-save is switched on. With a negative or zero prefix argument, auto-save is switched off. Using prefix arguments with Toggle Auto Save disregards the current state of auto-save.

Auto Save Toggle is a synonym for Toggle Auto Save.

auto-save is initially on or off in a new buffer according to the value of the editor variable default-auto-save-on.
3.5 File handling

**default-auto-save-on**  
*Editor Variable*

Default value: `t`

The default auto-save state of new buffers.

**auto-save-filename-pattern**  
*Editor Variable*

Default value: "~A#~A#"

This is a `format` control string used to make the filename of the checkpoint file. `format` is called with two arguments, the first being the directory namestring and the second being the file namestring of the default buffer pathname.

The default value causes the auto-save file to be created in the same directory as the file for which it is a backup, and with the name surrounded by `#` characters.

**auto-save-key-count-threshold**  
*Editor Variable*

Default value: 256

Specifies the number of destructive/modifying keystrokes that automatically trigger an auto-save of a buffer. If the value is `nil`, this feature is turned off.

**auto-save-checkpoint-frequency**  
*Editor Variable*

Default value: 300

Specifies the time interval in seconds after which all modified buffers which are in "Save" mode are auto-saved. If the value is `nil`, zero or negative, this feature is turned off.

**auto-save-cleanup-checkpoints**  
*Editor Variable*

Default value: `t`. 
This variable controls whether an auto-save function will cleanup by deleting the checkpoint file for a buffer after it is saved. If the value is true then this cleanup will occur.

### 3.5.4 Backing-up files on saving

When a file is explicitly saved in the editor, a backup is automatically made by writing the old contents of the file to a backup before saving the new version of the file. The backup file appears in the same directory as the original file. By default its name is the same as the original file followed by a \~ character.

**backups-wanted**  
*Editor Variable*

Default value: \texttt{t}  
Controls whether to make a backup copy of a file the first time it is modified. If the value is \texttt{t}, a backup is automatically made on first saving. If the value is \texttt{nil}, no backup is made.

**backup-filename-suffix**  
*Editor Variable*

Default value: \texttt{#\~}  
This variable contains the character used as a suffix for backup files. By default, this is the tilde (\~) character.

**backup-filename-pattern**  
*Editor Variable*

Default value: \\
"\~A\~A\~A"

This control string is used with the Common Lisp \texttt{format} function to create the filename of the backup file. \texttt{format} is called with three arguments, the first being the directory name-string and the second being the file name-string of the pathname associated with the buffer. The third is the value of the editor variable \texttt{backup-filename-suffix}.

The backup file is created in the same directory as the file for which it is a backup, and it has the same name, followed by the \texttt{backup-filename-suffix}. 
3.5 File handling

Note that the backup-suffix can be changed functionally as well as by interactive means. For example, the following code changes the suffix to the @ character:

```
(setf (editor:variable-value 'editor:backup-filename-suffix :current nil) 
"#@")
```

3.5.5 Miscellaneous file operations

**Print File**

*Editor Command*

Arguments: `file`

Key sequence: None


**Revert Buffer**

*Editor Command*

Arguments: None

Key sequence: None

If the current buffer is associated with a file, its contents revert to the state when it was last saved. If the buffer is not associated with a file, it is not possible for a previous state to be recovered.

If auto-save is on for the current buffer, the version of the file that is recovered is either that derived by means of an automatic save or by means of an explicit save, whichever is the most recent. If auto-save is off, the buffer reverts to its state when last explicitly saved.

If the buffer has been modified and the value of the variable `revert-buffer-confirm` is `t` then `Revert Buffer` asks for confirmation before reverting to a previous state.

Any prefix argument forces `Revert Buffer` to use the last explicitly saved version.
revert-buffer-confirm

Editor Variable

Default value: \texttt{t}

When the command \texttt{Revert Buffer} is invoked, if the value of this variable is \texttt{t} and the buffer has been modified then confirmation is requested before the revert operation is performed. If its value is \texttt{nil}, no confirmation is asked for.

Process File Options

Editor Command

Arguments: None
Key sequence: None

The attribute line at the top of the file is reprocessed, as if the file had just been read from disk. If no major mode is specified in the attribute line, the type of the file is used to determine the major mode. See “Modes” on page 91.

Insert File

Editor Command

Arguments: \texttt{pathname}
Key sequence: \texttt{Ctrl+X I \texttt{pathname}}

\texttt{editor:insert-file-command \texttt{p \&optional \texttt{pathname} \texttt{buffer}}}

Inserts the file defined by \texttt{pathname} into the current buffer at the current point.

The argument \texttt{buffer} is the buffer in which the file is to be inserted.

Delete File

Editor Command

Arguments: \texttt{pathname}
Key sequence: None

Deletes the file defined by \texttt{pathname}. The user is asked for confirmation before the file is deleted.
Delete File and Kill Buffer

Arguments: buffer
Key sequence: None

`editor:delete-file-and-kill-buffer-command p &optional buffer`

After confirmation from the user, this deletes the file associated with `buffer` and then kills the buffer.

Rename File

Arguments: file new-file-name
Key sequence: None

Changes the name of `file` to `new-file-name`.

If you are currently editing the file to be renamed, the buffer remains unaltered, retaining the name associated with the old file even after renaming has taken place. If you then save the current buffer, it is saved to a file with the name of the buffer, that is, to a file with the old name.

Make Directory

Arguments: None
Key sequence: None

Prompts the user for a directory name and makes it in the filesystem.

The prefix argument is ignored.

3.6 Movement

This section gives details of commands used to move the current point (indicated by the cursor) around the buffer.

The use of prefix arguments with this set of commands can be very useful, as they allow you to get where you want to go faster. In general, using a negative prefix argument repeats these commands a certain number of times in the opposite logical direction. For example, the command `Ctrl+U 10 Ctrl+B` moves the
cursor 10 characters backwards, but the command Ctrl+U -10 Ctrl+B moves the cursor 10 characters forward.

Some movement commands may behave slightly differently in different modes as delimiter characters may vary.

**Forward Character**  
*Editor Command*

Arguments: None  
Key sequence: Ctrl+F or Right Arrow on some keyboards  
Moves the current point forward one character.

**Backward Character**  
*Editor Command*

Arguments: None  
Key sequence: Ctrl+B or Left Arrow on some keyboards  
Moves the current point backward one character.

**Forward Word**  
*Editor Command*

Arguments: None  
Key sequence: Meta+F  
Moves the current point forward one word.

**Backward Word**  
*Editor Command*

Arguments: None  
Key sequence: Meta+B  
Moves the current point backward one word.

**Beginning of Line**  
*Editor Command*

Arguments: None  
Key sequence: Ctrl+A  
Moves the current point to the beginning of the current line.
3.6 Movement

**End of Line**  
*Editor Command*

Arguments: None  
Key sequence: **Ctrl+E**  
Moves the current point to the end of the current line.

**Next Line**  
*Editor Command*

Arguments: None  
Key sequence: **Ctrl+N** or Down Arrow on some keyboards  
Moves the current point down one line. If that would be after the end of the line, the current point is moved to the end of the line instead.

**Previous Line**  
*Editor Command*

Arguments: None  
Key sequence: **Ctrl+P** or Up Arrow on some keyboards  
Moves the current point up one line. If that would be after the end of the line, the current point is moved to the end of the line instead.

**Goto Line**  
*Editor Command*

Arguments: *number*  
Key sequence: None  
Moves to the line numbered *number*.

**What Line**  
*Editor Command*

Arguments: None.  
Key sequence: None  
Prints in the Echo Area the line number of the current point.
3 Command Reference

**Forward Sentence**

*Editor Command*

- Arguments: None
- Key sequence: Meta+E

Moves the current point to the end of the current sentence. If the current point is already at the end of a sentence, it is moved to the end of the next sentence.

**Backward Sentence**

*Editor Command*

- Arguments: None
- Key sequence: Meta+A

Moves the current point to the start of the current sentence. If the current point is already at the start of a sentence, it is moved to the beginning of the previous sentence.

**Forward Paragraph**

*Editor Command*

- Arguments: None
- Key sequence: Meta+]

Moves the current point to the end of the current paragraph. If the current point is already at the end of a paragraph, then it is moved to the end of the next paragraph.

**Backward Paragraph**

*Editor Command*

- Arguments: None
- Key sequence: Meta+[!

Moves the current point to the start of the current paragraph. If the current point is already at the start of a paragraph, then it is moved to the beginning of the previous paragraph.

**Scroll Window Down**

*Editor Command*

- Arguments: None
- Key sequence: Ctrl+V
3.6 Movement

editor:scroll-window-down-command p &optional window

Changes the text that is being displayed to be one screenful forward, minus scroll-overlap. If the current point is no longer included in the new text, it is moved to the start of the line nearest to the centre of the window.

A prefix argument causes the current screen to be scrolled up the number of lines specified and that number of new lines are shown at the bottom of the window.

The argument window is the name of the window to be scrolled. The default is the current window.

Scroll Window Up Editor Command

Arguments: None
Key sequence: Meta+V

editor:scroll-window-up-command p &optional window

Changes the text that is being displayed to be one screenful back, minus scroll-overlap. If the current point is no longer included in the new text, it is moved to the start of the line nearest to the centre of the window.

A prefix argument causes the current screen to be scrolled down the number of lines specified and that number of new lines are shown at the top of the window.

The argument window is the name of the window to be scrolled. The default is the current window.

scroll-overlap Editor Variable

Default value: 1

Determines the number of lines of overlap when Scroll Window Down and Scroll Window Up are used with no prefix argument.
Line to Top of Window  
**Editor Command**
- Arguments: None
- Key sequence: None
- Moves the current line to the top of the window.

Top of Window  
**Editor Command**
- Arguments: None
- Key sequence: None
- Moves the current point to the start of the first line currently displayed in the window.

Bottom of Window  
**Editor Command**
- Arguments: None
- Key sequence: None
- Moves the current point to the start of the last line that is currently displayed in the window.

Move to Window Line  
**Editor Command**
- Arguments: None
- Key sequence: Meta+Shift+R
- Without a prefix argument, moves the current point to the start of the center line in the window.
- With a positive (negative) integer prefix argument \( p \), moves the point to the start of the \( p \)th line from the top (bottom) of the window.

Beginning of Buffer  
**Editor Command**
- Arguments: None
- Key sequence: Meta+Shift+<
- Moves the current point to the beginning of the current buffer.
3.6 Movement

**End of Buffer**  
*Editor Command*

Arguments: None  
Key sequence: Meta+Shift+>

Moves the current point to the end of the current buffer.

**Skip Whitespace**  
*Editor Command*

Arguments: None  
Key sequence: None

Skips to the next non-whitespace character if the current character is a whitespace character (for example, *Space*, *Tab* or newline).

**What Cursor Position**  
*Editor Command*

Arguments: None  
Key sequence: Ctrl+X =

Displays in the echo area the character under the point and the column of the point. Also available via the function:  

**Where Is Point**  
*Editor Command*

Arguments: None  
Key sequence: None

Displays in the echo area the position of the current point in terms of characters in the buffer, as a fraction of current point position over total buffer length.

**Goto Point**  
*Editor Command*

Arguments: `point`  
Key sequence: None

Moves the current point to `point`, where `point` is a character position in the current buffer.
3.7 Marks and regions

The first part of this section gives details of commands associated with marking, while the second provides details of a few commands whose area is limited to a region. Other region specific commands are available but are dealt with in more appropriate sections of this manual. For example, Write Region is dealt with under the “File handling” on page 25 as it involves writing a region to a file.

Details of marks are kept in a mark ring so that previously defined marks can be accessed. The mark ring works like a stack, in that marks are pushed onto the ring and can only be popped off on a "last in first out" basis. Each buffer has its own mark ring.

Note that marks may also be set by using the mouse—see “Buffers, windows and the mouse” on page 113—but also note that a region must be defined either by using the mouse or by using editor key sequences, as the region may become unset if a combination of the two is used. For example, using Ctrl+Space to set a mark and then using the mouse to go to the start of the required region unsets the mark.

3.7.1 Marks

Set Mark

Arguments: None
Key sequence: Ctrl+Space or Middle Mouse Button

With no prefix argument, pushes the current point onto the mark ring, effectively setting the mark to the current point, and makes the activates the region.

With a prefix argument equal to the value of the prefix-argument-default, Pop and Goto Mark is invoked.

With a prefix argument equal to the square of the prefix-argument-default (achieved by typing Ctrl+U Ctrl+U before invoking Set Mark), Pop Mark is invoked.
3.7 Marks and regions

**Pop and Goto Mark**

*Editor Command*

Arguments: None
Key sequence: None

Moves the current point to the mark without saving the current point on
the mark ring (in contrast with Exchange Point and Mark). After the cur-
rent point has been moved to the mark, the mark ring is rotated. The cur-
rent region is de-activated.

**Pop Mark**

*Editor Command*

Arguments: None
Key sequence: *Meta*+Ctrl+Space

Rotates the mark ring so that the previous mark becomes the current
mark. The point is not moved but the current region is de-activated.

**Exchange Point and Mark**

*Editor Command*

Arguments: None
Key sequence: *Ctrl*+X *Ctrl*+X

`editor:exchange-point-and-mark-command p &optional buffer`

Sets the mark to the current point and moves the current point to the pre-
vious mark. This command can therefore be used to examine the extent of
the current region.

The argument *buffer* is the buffer in which to exchange the point and mark.
The default value is the current buffer.

**Mark Word**

*Editor Command*

Arguments: *number*
Key sequence: *Meta*+@

Marks the word following the current point. A prefix argument, if sup-
plied, specifies the number of words marked.
3 Command Reference

Mark Sentence  Editor Command

Arguments: None
Key sequence: None

Puts the mark at the end of the current sentence and the current point at the start of the current sentence. The sentence thereby becomes the current region. If the current point is initially located between two sentences then the mark and current point are placed around the next sentence.

Mark Paragraph  Editor Command

Arguments: None
Key sequence: Meta+H

Puts the mark at the end of the current paragraph and the current point at the start of the current paragraph. The paragraph thereby becomes the current region. If the current point is initially located between two paragraphs, then the mark and current point are placed around the next paragraph.

Mark Whole Buffer  Editor Command

Arguments: None
Key sequence: Ctrl+X H

Sets the mark at the end of the current buffer and the current point at the beginning of the current buffer. The current region is thereby set as the whole of the buffer.

A non-nil prefix argument causes the mark to be set as the start of the buffer and the current point at the end.

3.7.2 Regions

Count Words Region  Editor Command

Arguments: None
Key sequence: None
Displays a count of the total number of words in the region between the current point and the mark.

**Count Lines Region**  
*Editor Command*  
Arguments: None  
Key sequence: None  
Displays a count of the total number of lines in the region between the current point and the mark.

**region-query-size**  
*Editor Variable*  
Default value: 60  
If the region between the current point and the mark contains more lines than the value of this editor variable, then any destructive operation on the region prompts the user for confirmation before being executed.

**Print Region**  
*Editor Command*  
Arguments: None  
Key sequence: None  
Prints the current region, using `capi:print-text`. See the *LispWorks CAPI Reference Manual* for details of this function.

### 3.8 Locations

A *location* is the position of the current point in a buffer at some time in the past. Locations are recorded automatically by the editor for most commands that take you to a different buffer or where you might lose your place within the current buffer (for example *Beginning of Buffer*). They are designed to be a more comprehensive form of the mark ring (see *Pop and Goto Mark*), but without the interaction with the selected region.
3 Command Reference

Go Back  Editor Command

Arguments: None
Key sequence: Ctrl+X C

Takes you back to the most recently recorded location. If a prefix argument count is supplied, it takes you back count locations in the location history. If count is negative, it takes you forward again count locations in the history, provided that no more locations have been recorded since you last went back.

Select Go Back  Editor Command

Arguments: None
Key sequence: Ctrl+X M

Takes you back to a previously recorded location, which you select from a list.

Any prefix argument is ignored.

Go Forward  Editor Command

Arguments: None
Key sequence: Ctrl+X P

Takes you back to the next location in the ring of recorded locations. If a prefix argument count is supplied, it takes you forward count locations in the location history. If count is negative, it takes you back count locations in the history.

3.9 Deleting and killing text

There are two ways of removing text: deletion, after which the deleted text is not recoverable (except with the undo command); and killing, which appends the deleted text to the kill ring, so that it may be recovered using the Un-Kill and Rotate Kill Ring commands. The first section contains details of commands to delete text, and the second details of commands to kill text.
Note that, if Delete Selection Mode is active, then any currently selected text is deleted when text is entered. See ‘Delete Selection’ on page 3-56 for details.

The use of prefix arguments with this set of commands can be very useful. In general, using a negative prefix argument repeats these commands a certain number of times in the opposite logical direction. For example, the key sequence Ctrl+U 10 Meta+D deletes 10 words after the current point, but the key sequence Ctrl+U -10 Meta+D deletes 10 words before the current point.

### 3.9.1 Deleting Text

#### Delete Next Character

*Editor Command*

- **Arguments:** None
- **Key sequence:** Ctrl+D
- Deletes the character immediately after the current point.

#### Delete Previous Character

*Editor Command*

- **Arguments:** None
- **Key sequence:** Delete
- Deletes the character immediately before the current point.

#### Delete Previous Character Expanding Tabs

*Editor Command*

- **Arguments:** None
- **Key sequence:** None
- Deletes the character immediately before the current point, but if the previous character is a Tab, then this is expanded into the equivalent number of spaces, so that the apparent space is reduced by one.

A prefix argument deletes the required number of characters, but if any of them are tabs, the equivalent spaces are inserted before the deletion continues.
Delete Horizontal Space  
*Editor Command*

Arguments: None  
Key sequence: Meta+\  
Deletes all spaces on the line surrounding the current point.

Just One Space  
*Editor Command*

Arguments: None  
Key sequence: Meta+Space  
Deletes all space on the current line surrounding the current point and then inserts a single space. If there was initially no space around the current point, a single space is inserted.

Delete Blank Lines  
*Editor Command*

Arguments: None  
Key sequence: Ctrl+X Ctrl+O  
If the current point is on a blank line, all surrounding blank lines are deleted, leaving just one. If the current point is on a non-blank line, all following blank lines up to the next non-blank line are deleted.

Delete Region  
*Editor Command*

Arguments: None  
Key sequence: None  
Delete the current region. Also available via `editor:delete-region-command`.

Clear Listener  
*Editor Command*

Arguments: None  
Key sequence: None
3.9 Deleting and killing text

Deletes the text in a Listener, leaving you with a prompt. Undo information is not retained, although you are warned about this before confirming the command.

This command is useful if the Listener session has grown very large.

**Clear Output**

*Editor Command*

Arguments: None
Key sequence: None

Deletes the text in the Output tab of a Listener or Editor tool, or an Output Browser. Undo information is discarded without warning.

This command is useful if the output has grown very large.

3.9.2 Killing text

Most of these commands result in text being pushed onto the kill ring so that it can be recovered. There is only one kill ring for all buffers so that text can be copied from one buffer to another.

Normally each kill command pushes a new block of text onto the kill ring. However, if more than one kill command is issued sequentially, and the text being killed was next to the previously killed text, they form a single entry in the kill ring (exceptions being Kill Region and Save Region).

Append Next Kill is different in that affects where a subsequent killed text is stored in the kill ring, but does not itself modify the kill ring.

**Kill Next Word**

*Editor Command*

Arguments: None
Key sequence: Meta+D

Kills the rest of the word after the current point. If the current point is between two words, then the next word is killed.
**Kill Previous Word**  
*Editor Command*

- Arguments: None
- Key sequence: `Meta+Backspace`

Kills the rest of the word before the current point. If the current point is between two words, then the previous word is killed.

**Kill Line**  
*Editor Command*

- Arguments: None
- Key sequence: `Ctrl+K`

Kills the characters from the current point up to the end of the current line. If the line is empty then the line is deleted.

**Backward Kill Line**  
*Editor Command*

- Arguments: None
- Key sequence: None

Kills the characters from the current point to the beginning of the line. If the current point is already at the beginning of the line, the current line is joined to the previous line, with any trailing space on the previous line killed.

**Forward Kill Sentence**  
*Editor Command*

- Arguments: None
- Key sequence: `Meta+K`

Kills the text starting from the current point up to the end of the sentence. If the current point is between two sentences, then the whole of the next sentence is killed.

**Backward Kill Sentence**  
*Editor Command*

- Arguments: None
- Key sequence: `Ctrl+X Backspace`
Kills the text starting from the current point up to the beginning of the sentence. If the current point is between two sentences, then the whole of the previous sentence is killed.

**Kill Region** *Editor Command*

Arguments: None
Key sequence: `Ctrl+W`
Kills the region between the current point and the mark.

**Save Region** *Editor Command*

Arguments: None
Key sequence: `Meta+W`
Pushes the region between the current point and the mark onto the kill ring without deleting it from the buffer. Text saved in this way can therefore be inserted elsewhere without first being killed.

**Append Next Kill** *Editor Command*

Arguments: None
Key sequence: `Meta+Ctrl+W`
If the next command entered kills any text then this text will be appended to the existing kill text instead of being pushed separately onto the kill ring.

### 3.10 Inserting text

This section contains details of commands used to insert text from the kill ring—see “Deleting and killing text” on page 48—and various other commands used to insert text and lines into the buffer.

**Un-Kill** *Editor Command*

Arguments: None
Key sequence: `Ctrl+Y`
Selects (yanks) the top item in the kill ring (which represents the last piece of text that was killed with a kill command or saved with Save Region) and inserts it before the current point. The current point is left at the end of the inserted text, and the mark is automatically set to the beginning of the inserted text.

A prefix argument (Ctrl+U number) causes the item at position number in the ring to be inserted. The order of items on the ring remains unaltered.

**Rotate Kill Ring**

*Editor Command*

Arguments: None

Key sequence: Meta+Y

Replaces the text that has just been un-killed with the item that is next on the kill ring. It is therefore possible to recover text other than that which was most recently killed by typing Ctrl+Y followed by Meta+Y the required number of times. If Un-Kill was not the previous command, an error is signalled.

Note that the ring is only rotated and no items are actually deleted from the ring using this command.

A prefix argument causes the kill ring to be rotated the appropriate number of times before the top item is selected.

**New Line**

*Editor Command*

Arguments: None

Key sequence: Return

Opens a new line before the current point. If the current point is at the start of a line, an empty line is inserted above it. If the current point is in the middle of a line, that line is split. The current point always becomes located on the second of the two lines.

A prefix argument causes the appropriate number of lines to be inserted before the current point.
3.10 Inserting text

Open Line

*Editor Command*

Arguments: None
Key sequence: Ctrl+O

Opens a new line after the current point. If the current point is at the start of a line, an empty line is inserted above it. If the current point is in the middle of a line, that line is split. The current point always becomes located on the first of the two lines.

A prefix argument causes the appropriate number of lines to be inserted after the current point.

Quoted Insert

*Editor Command*

Arguments: *args*
Key sequence: Ctrl+Q &rest *args*

*Quoted Insert* is a versatile command allowing you to enter characters which are not accessible directly on your keyboard.

A single argument *key* is inserted into the text literally. This can be used to enter control keys (such as Ctrl+L) into a buffer as a text string. Note that Ctrl is represented by ^ and Meta by \^.

You may input a character by entering its Octal Unicode code: press Return to indicate the end of the code. For example enter

Ctrl+Q 4 3 Return

to input #.

Self Insert

*Editor Command*

Arguments: None
Key sequence: *key*

*editor:self-insert-command* *p* &optional *char*

This is the basic command used for inserting each character that is typed. The character to be inserted is *char*. There is no need for the user to use this command explicitly.
Dynamic Completion  
*Editor Command*

Arguments: None  
Key sequence: Meta+/

Tries to complete the current word, by looking backwards for a word that starts with the same characters as have already been typed. Repeated use of this command makes the search skip to successively previous instances of words beginning with these characters. A prefix argument causes the search to progress forwards rather than backwards. If the buffer is in Lisp mode then completion occurs for Lisp symbols as well as words.

Expand File Name  
*Editor Command*

Arguments: None  
Key sequence: Meta+Tab

Expands the file name at the current point. Issuing this command twice in succession brings up a list of possible completions in a popup window.

3.11 Delete Selection

When in Delete Selection Mode, commands that insert text into the buffer first delete any selected text. Delete Selection Mode is a global editor setting. It is off by default with Emacs keys, and is on by default when using KDE/Gnome editor emulation.

Delete Selection Mode  
*Editor Command*

Arguments: None  
Key Sequence: None

Toggles Delete Selection Mode, switching it on if it is currently off, and off if it is currently on.

3.12 Undoing

Commands that modify the text in a buffer can be undone, so that the text reverts to its state before the command was invoked, using Undo. Details of
modifying commands are kept in an undo ring so that previous commands can be undone. The undo ring works like a stack, in that commands are pushed onto the ring and can only be popped off on a "last in first out" basis. un-Kill can also be used to replace text that has inadvertently been deleted.

**Undo**

*Editor Command*

Arguments: None

Key sequence: Ctrl+Shift+_

Undoes the last command. If typed repeatedly, the most recent commands in the editing session are successively undone.

**undo-ring-size**

*Editor Variable*

Default value: 100

The number of items in the undo ring.

### 3.13 Case conversion

This section provides details of the commands which allow case conversions on both single words and regions of text. The three general types of case conversion are converting words to uppercase, converting words to lowercase and converting the first letter of words to uppercase.

**Lowercase Word**

*Editor Command*

Arguments: None

Key sequence: Meta+L

Converts the current word to lowercase, starting from the current point. If the current point is between two words, then the next word is converted.

A negative prefix argument converts the appropriate number of words before the current point to lowercase, but leaves the current point where it was.
3 Command Reference

**Uppercase Word**

*Editor Command*

Arguments: None

Key sequence: Meta+U

Converts the current word to uppercase, starting from the current point. If the current point is between two words, then the next word is converted.

A negative prefix argument converts the appropriate number of words before the current point to uppercase, but leaves the current point where it was.

**Capitalize Word**

*Editor Command*

Arguments: None

Key sequence: Meta+C

Converts the current word to lowercase, capitalizing the first character. If the current point is inside a word, the character immediately after the current point is capitalized.

A negative prefix argument capitalizes the appropriate number of words before the current point, but leaves the point where it was.

**Lowercase Region**

*Editor Command*

Arguments: None

Key sequence: Ctrl+X Ctrl+L

Converts all the characters in the region between the current point and the mark to lowercase.

**Uppercase Region**

*Editor Command*

Arguments: None

Key sequence: Ctrl+X Ctrl+U

Converts all the characters in the region between the current point and the mark to uppercase.
Capitalize Region

Arguments: None
Key sequence: None

Converts all the words in the region between the mark and the current point to lowercase, capitalizing the first character of each word.

3.14 Transposition

This section gives details of commands used to transpose characters, words, lines and regions.

Transpose Characters

Arguments: None
Key sequence: Ctrl+T

Transposes the current character with the previous character, and then moves the current point forwards one character.

If this command is issued when the current point is at the end of a line, the two characters to the left of the cursor are transposed.

A positive prefix argument causes the character before the current point to be shifted forwards the required number of places. A negative prefix argument has a similar effect but shifts the character backwards. In both cases the current point remains located after the character which has been moved.

Transpose Words

Arguments: None
Key sequence: Meta+T

Transposes the current word with the next word, and then moves the current point forward one word. If the current point is initially located between two words, then the previous word is moved over the next word.

A positive prefix argument causes the current or previous word to be shifted forwards the required number of words. A negative prefix argu-
ment has a similar effect but shifts the word backwards. In both cases the current point remains located after the word which has been moved.

**Transpose Lines**  
*Editor Command*

Arguments: None  
Key sequence: `Ctrl+X Ctrl+T`

Transposes the current line with the previous line, and then moves the current point forward one line.

A positive prefix argument causes the previous line to be shifted forwards the required number of lines. A negative prefix argument has a similar effect but shifts the line backwards. In both cases the current point remains located after the line which has been moved.

A prefix argument of zero transposes the current line and the line containing the mark.

**Transpose Regions**  
*Editor Command*

Arguments: None  
Key sequence: None

Transposes two regions. One region is delineated by the current point and the mark. The other region is delineated by the next two points on the mark ring. To use this command it is necessary to use **Set Mark** at the beginning and end of one region and at the beginning of the other region, and then move the current point to the end of the second region.

### 3.15 Overwriting

In the default mode of operation, each character that is typed is inserted into the text, with the existing characters being shifted as appropriate. In overwrite mode, each character that is typed deletes an existing character in the text.

When in overwrite mode, a character can be inserted without deleting an existing character by preceding it with `Ctrl+Q`.  

3.16 Indentation

**Overwrite Mode**  
*Editor Command*

Arguments: None  
Key sequence: **Insert**

Switches overwrite mode on if it is currently off, and off if it is currently on.

With a positive prefix argument, overwrite mode is turned on. With a zero or negative prefix argument it is turned off. Using prefix arguments with **Overwrite Mode** disregards the current state of the mode.

**Self Overwrite**  
*Editor Command*

Arguments: None  
Key sequence: **key**

If the current point is in the middle of a line, the next character (that is, the character that is highlighted by the cursor) is replaced with the last character typed. If the current point is at the end of a line, the new character is inserted without removing any other character.

A prefix argument causes the new character to overwrite the relevant number of characters.

This is the command that is invoked when each character is typed in overwrite mode. There is no need for users to invoke this command explicitly.

**Overwrite Delete Previous Character**  
*Editor Command*

Arguments: None  
Key sequence: None

Replaces the previous character with space, except that tabs and newlines are deleted.

### 3.16 Indentation

This section contains details of commands used to indent text. Indentation is usually achieved by inserting tab or space characters into the text so as to indent that text a predefined number of spaces.
The effect of the editor indentation commands depends on the major mode of the buffer. Where relevant, the command details given below provide information on how they operate in Text mode and Lisp mode. The operation of commands in Fundamental mode is generally the same as that of Text mode.

**Indent**

*Editor Command*

Arguments: None
Key sequence: `Tab`

In Text mode, `spaces-for-tab #\Space` characters are inserted. A prefix argument causes this to occur at the start of the appropriate number of lines (starting from the current line).

In Lisp mode, the current line is indented according to the structure of the current Lisp form. A prefix argument `p` causes `p` lines to be indented according to Lisp syntax.

See `editor:*indent-with-tabs*` for control over the insertion of `#\Tab` characters by this and other indentation commands.

**Note:** the key sequence `Tab` is overridden in Lisp mode to perform *Indent Selection* or *Complete Symbol*.

**spaces-for-tab**

*Editor Variable*

Default value: 8

Determines the width of the whitespace (that is, the number of `#\Space` characters) used to display a `#\Tab` character.

**Indent Region**

*Editor Command*

Arguments: None
Key sequence: `Meta+Ctrl+\`

Indents all the text in the region between the mark and the current point.

In Text mode a block of whitespace, which is `spaces-for-tab` wide, is inserted at the start of each line within the region.

In Lisp mode the text is indented according to the syntax of the Lisp form.
In both cases, a prefix argument causes any existing indentation to be deleted and replaced with a block of whitespace of the appropriate width.

**Indent Rigidly**

*Editor Command*

Arguments: None

Key sequence: Ctrl+X Tab or Ctrl+X Ctrl+I

Indents each line in the region between the current point and the mark by a block of whitespace which is spaces-for-tab wide. Any existing whitespace at the beginning of the lines is retained.

A positive prefix argument causes the lines to be indented by the appropriate number of spaces, in addition to their existing space. A negative prefix argument causes the lines to be shifted to the left by the appropriate number of spaces. Where necessary, tabs are converted to spaces.

**Indent Selection**

*Editor Command*

Arguments: None

Key sequence: None

Indents all the text in the selection or the current line if there is no selection. With a prefix argument p, any existing indentation is deleted and replaced with a block of space p columns wide.

See also Indent Selection or Complete Symbol.

**Delete Indentation**

*Editor Command*

Arguments: None

Key sequence: Meta+Shift+`

Joins the current line with the previous one, deleting all whitespace at the beginning of the current line and at the end of the previous line. The deleted whitespace is normally replaced with a single space. However, if the deleted whitespace is at the beginning of a line, or immediately after a `, or immediately before a `, then the whitespace is merely deleted without any characters being inserted. If the preceding character is a sentence terminator, then two spaces are left instead of one.
A prefix argument causes the following line to be joined with the current line.

**Back to Indentation**  
*Editor Command*  
Arguments: None  
Key sequence: Meta+M  
Moves the current point to the first character in the current line that is not a whitespace character.

**Indent New Line**  
*Editor Command*  
Arguments: None  
Key sequence: None  
Moves everything to the right of the current point to a new line and indents it. Any whitespace before the current point is deleted. If there is a fill-prefix, this is inserted at the start of the new line instead.  
A prefix argument causes the current point to be moved down the appropriate number of lines and indented.

**Quote Tab**  
*Editor Command*  
Arguments: None  
Key sequence: None  
Inserts a Tab character.  
A prefix argument causes the appropriate number of tab characters to be inserted.

### 3.17 Filling

Filling involves re-formatting text so that each line extends as far to the right as possible without any words being broken or any text extending past the fill-column.
The first section deals with general commands used to fill text, while the second section provides information on Auto-Fill mode and related commands.

### 3.17.1 Fill commands

**Fill Paragraph**  
*Editor Command*

Arguments: None  
Key sequence: `Meta+Q`

Fills the current paragraph. If the current point is located between two paragraphs, the next paragraph is filled.

A prefix argument causes the current fill operation to use that value, rather than the value of `fill-column`.

**Fill Region**  
*Editor Command*

Arguments: None  
Key sequence: `Meta+G`

Fills the region from the current point to the mark.

A prefix argument causes the current fill operation to use that value, rather than the value of `fill-column`.

**fill-column**  
*Editor Variable*

Default value: 70

Determines the column at which text in the current buffer is forced on to a new line when filling text.

**Set Fill Column**  
*Editor Command*

Arguments: None  
Key sequence: `Ctrl+X F`

Sets the value of `fill-column`, for the current buffer, as the column of the current point.
A prefix argument causes `fill-column` to be set at the required value.

**fill-prefix**

*Editor Variable*

Default value: `nil`

Defines a string which is excluded when each line of the current buffer is re-formatted using the filling commands. For example, if the value is `";;;"`, then these characters at the start of a line are skipped over when the text is re-formatted. This allows you to re-format (fill) Lisp comments. If the value is `nil`, no characters are excluded when text is filled.

If the values is non-nil, any line that does not begin with the value is considered to begin a new paragraph. Therefore, any re-formatting of comments in Lisp code does not intrude outside the commented lines.

**Set Fill Prefix**

*Editor Command*

Arguments: None

Key sequence: `Ctrl+X .`

Sets the `fill-prefix` of the current buffer to be the text from the beginning of the current line up to the current point. The `fill-prefix` may be set to `nil` by using this command with the current point at the start of a line.

**Center Line**

*Editor Command*

Arguments: None

Key sequence: None

Centers the current line with reference to the current value of `fill-column`.

A prefix argument causes the current line to be centered with reference to the required width.
3.17 Filling

3.17.2 Auto-fill mode

In the default mode of operation, no filling of text takes place unless specified by using one of the commands described above. A result of this is that the user has to press `Return` at the end of each line typed to simulate filling. In Auto-Fill mode lines are broken between words at the right margin automatically as the text is being typed. Each line is broken when a space is inserted, and the text that extends past the right margin is put on the next line. The right hand margin is determined by the editor variable `fill-column`.

**Auto Fill Mode**

*Editor Command*

Arguments: None  
Key sequence: None  
Switches auto-fill mode on if it is currently off, and off if it is currently on.  
With a positive prefix argument, auto-fill mode is switched on. With a negative or zero prefix argument, it is switched off. Using prefix arguments with **Auto Fill Mode** disregards the current state of the mode.

**Auto Fill Space**

*Editor Command*

Arguments: None  
Key sequence: `Space`  
Mode: Auto-Fill  
Inserts a space and breaks the line between two words if the line extends beyond the right margin. A fill prefix is automatically added at the beginning of the new line if the value of `fill-prefix` is non-nil.  
When `Space` is bound to this command in Auto-Fill mode, this key no longer invokes **Self Insert**.  
A positive prefix argument causes the required number of spaces to be inserted but no line break. A prefix argument of zero causes a line break, if necessary, but no spaces are inserted.
3 Command Reference

**Auto Fill Linefeed**  
*Editor Command*

Arguments: None  
Key sequence: `Linefeed`  
Mode: Auto-Fill

Inserts a `Linefeed` and a `fill-prefix` (if one exists).

**Auto Fill Return**  
*Editor Command*

Arguments: None  
Key sequence: `Return`  
Mode: Auto-Fill

The current line is broken, between two words if necessary, with no Space being inserted. This is equivalent to `Auto Fill Space` with a zero prefix argument, but followed by a newline.

**auto-fill-space-indent**  
*Editor Variable*

Default value: `nil`

When true, Auto-fill commands use `Indent New Comment Line` to break lines instead of `New Line`.

3.18 Buffers

This section contains details of commands used to manipulate buffers.

**Select Buffer**  
*Editor Command*

Arguments: `buffer-name`  
Key sequence: `Ctrl+X B buffer-name`

Displays a buffer called `buffer-name` in the current window. If no buffer name is provided, the last buffer accessed in the current window is displayed. If the buffer that is selected is already being displayed in another window, any modifications to that buffer are shown simultaneously in both windows.
Select Buffer Other Window  

*Editor Command*

Arguments: `buffer-name`

Key sequence: None

Displays a buffer called `buffer-name` in a new window. If no buffer name is provided, the last buffer displayed in the current window is selected. If the buffer that is selected is already being displayed in another window, any modifications to that buffer are shown simultaneously in both windows.

Select Previous Buffer  

*Editor Command*

Arguments: None

Key sequence: `Meta+Ctrl+L`

Displays the last buffer accessed in a new window. If the buffer that is selected is already being displayed in another window, any modifications to that buffer are shown simultaneously in both windows.

A prefix argument causes the appropriately numbered buffer, from the top of the buffer history, to be selected.

Circulate Buffers  

*Editor Command*

Arguments: None

Key sequence: `Meta+Shift+Ctrl+L`

Move through the buffer history, selecting the successive previous buffers.

Kill Buffer  

*Editor Command*

Arguments: `buffer-name`

Key sequence: `Ctrl+X K buffer-name`

**`editor:kill-buffer-command` p optional `buffer-name`**

Deletes a buffer called `buffer-name`. If no buffer name is provided, the current buffer is deleted. If the buffer that is selected for deletion has been modified then confirmation is asked for before deletion takes place.
List Buffers  Editor Command

Arguments: None
Key sequence: Ctrl+X Ctrl+B

Displays a list of all the existing buffers in the Buffers window in the Editor tool. Information is given on the name of the buffer, its mode, whether it has been modified or not, the pathname of any file it is associated with, and its size.

A buffer can be selected by clicking the left mouse button on the buffer name. The buttons on the toolbar can then be used to modify the selected buffer.

Create Buffer  Editor Command

Arguments: buffer-name
Key sequence: None

`editor:create-buffer-command p &optional buffer-name`

Creates a buffer called `buffer-name`. If no buffer name is provided then the current buffer is selected. If a buffer with the specified name already exists then this becomes the current buffer instead, and no new buffer is created.

New Buffer  Editor Command

Arguments: None
Key sequence: None

Creates a new unnamed buffer. The buffer is in Lisp mode.

default-buffer-element-type  Editor Variable

Default value: `lw:simple-char`

The character element type used when a new buffer is created, for example by New Buffer.
3.18 Buffers

Insert Buffer  Editor Command

Arguments: buffer-name
Key sequence: None

Inserts the contents of a buffer called buffer-name at the current point. If no buffer name is provided, the contents of the last buffer displayed in the current window are inserted.

Rename Buffer  Editor Command

Arguments: new-name
Key sequence: None

Changes the name of the current buffer to new-name.

Print Buffer  Editor Command

Arguments: None
Key sequence: None

Prints the current buffer, using capi:print-text. See the LispWorks CAPI Reference Manual for details of this function.

Toggle Buffer Read-Only  Editor Command

Arguments: None
Key sequence: Ctrl+X Ctrl+Q

Makes the current buffer read only, so that no modification to its contents are allowed. If it is already read only, this restriction is removed.

Check Buffer Modified  Editor Command

Arguments: None
Key sequence: Ctrl+X Shift+-

Checks whether the current buffer is modified or not.
Buffer Not Modified

**Editor Command**

Arguments: None
Key sequence: Meta+Shift+-

`editor:buffer-not-modified-command p &optional buffer`

Makes the current buffer not modified.
The argument `buffer` is the name of the buffer to be un-modified. The default is the current buffer.

3.19 Windows

This section contains details of commands used to manipulate windows. A window ring is used to hold details of all windows currently open.

**New Window**

**Editor Command**

Arguments: None
Key sequence: Ctrl+X 2

Creates a new window and makes it the current window. Initially, the new window displays the same buffer as the current one.

**Next Window**

**Editor Command**

Arguments: None
Key sequence: None

Changes the current window to be the next window in the window ring, and the current buffer to be the buffer that is displayed in that window.

**Next Ordinary Window**

**Editor Command**

Arguments: None
Key sequence: Ctrl+X 0

Changes the current window to be the next ordinary editor window, thus avoiding the need to cycle through other window types (for example, Listeners and Debuggers).
Previous Window  

*Editor Command*

Arguments: None  
Key sequence: None

Changes the current window to be the previous window visited, and the current buffer to be the buffer that is displayed in that window.

Delete Window  

*Editor Command*

Arguments: None  
Key sequence: Ctrl+X 0

Deletes the current window. The previous window becomes the current window.

Delete Next Window  

*Editor Command*

Arguments: None  
Key sequence: Ctrl+X 1

Deletes the next window in the window ring.

Scroll Next Window Down  

*Editor Command*

Arguments: None  
Key sequence: None

The next window in the window ring is scrolled down.

A prefix argument causes the appropriately numbered window, from the top of the window ring, to be scrolled.

Scroll Next Window Up  

*Editor Command*

Arguments: None  
Key sequence: None

The next window in the window ring is scrolled up.
A prefix argument causes the appropriately numbered window, from the top of the window ring, to be scrolled.

**Toggle Count Newlines**

*Editor Command*

Arguments: None

Key sequence: None

Controls the size of the scroller in editor-based tools, and how the Editor tool’s mode line represents the extent of the displayed part of the buffer.

**Toggle Count Newlines** switches between counting newlines and counting characters in the current buffer. The counting determines what is displayed in the Editor tool’s mode line, and how the size of the scroller is computed.

When counting newlines, the mode line shows line numbers and the total number of lines:

\[
\text{StartLine} - \text{EndLine} \left[ \text{TotalLine} \right]
\]

When counting characters, the mode line shows percentages based on the characters displayed compared to the total number of characters in the buffer:

\[
\text{PercentStart} - \text{PercentEnd\%}
\]

The default behavior is counting newlines, except for very large buffers.

**Refresh Screen**

*Editor Command*

Arguments: None

Key sequence: Ctrl+L

Moves the current line to the center of the current window, and then re-displays all the text in all the windows.

A prefix argument of 0 causes the current line to become located at the top of the window. A positive prefix argument causes the current line to become located the appropriate number of lines from the top of the window. A negative prefix argument causes the current line to become located the appropriate number of lines from the bottom of the window.
3.20 Pages

Files are sometimes thought of as being divided into pages. For example, when a file is printed on a printer, it is divided into pages so that each page appears on a fresh piece of paper. The ASCII key sequence Ctrl+L constitutes a page delimiter (as it starts a new page on most line printers). A page is the region between two page delimiters. A page delimiter can be inserted into text being edited by using the editor command Quoted Insert (that is, type in Ctrl+Q Ctrl+L).

Previous Page

*Editor Command*

Arguments: None
Key sequence: Ctrl+X \[

Moves the current point to the start of the current page.

A prefix argument causes the current point to be moved backwards the appropriate number of pages.

Next Page

*Editor Command*

Arguments: None
Key sequence: Ctrl+X ]

Moves the current point to the start of the next page.

A prefix argument causes the current point to be moved forwards the appropriate number of pages.

Goto Page

*Editor Command*

Arguments: None
Key sequence: None

Moves the current point to the start of the next page.

A positive prefix argument causes the current point to be moved to the appropriate page starting from the beginning of the buffer. A negative prefix argument causes the current point to be moved back the appropriate number of pages from the current location. A prefix argument of zero
causes the user to be prompted for a string, and the current point is moved to the next page with that string contained in the page title.

Mark Page Editor Command

Arguments: None
Key sequence: Ctrl+X Ctrl+P

Puts the mark at the end of the current page and the current point at the start of the current page. The page thereby becomes the current region.

A prefix argument marks the page which is the appropriate number of pages on from the current one.

Count Lines Page Editor Command

Arguments: None
Key sequence: Ctrl+X L

Displays the number of lines in the current page and the location of the current point within the page.

A prefix argument displays the total number of lines in the current buffer and the location of the current point within the buffer (so that page delimiters are ignored).

View Page Directory Editor Command

Arguments: None
Key sequence: None

Displays a list of the first non-blank line after each page delimiter.

Insert Page Directory Editor Command

Arguments: None
Key sequence: None
Inserts a listing of the first non-blank line after each page delimiter at the start of the buffer, moving the current point to the end of this list. The location of the start of this list is pushed onto the mark ring.

A prefix argument causes the page directory to be inserted at the current point.

### 3.21 Searching and replacing

This section is divided into three parts. The first two provide details of commands used for searching. These commands are, on the whole, non-modifying and non-destructive, and merely search for strings and patterns. The third part provides details of commands used for replacing a string or pattern.

#### 3.21.1 Searching

Most of the search commands perform straightforward searches, but there are two useful commands (Incremental Search and Reverse Incremental Search) which perform incremental searches. This means that the search is started as soon as the first character is typed.

**Incremental Search**

*Editor Command*

**Arguments:** string  
**Key sequence:** Ctrl+S string

Searches forward, starting from the current point, for the search string that is input, beginning the search as soon as each character is typed in. When a match is found for the search string, the current point is moved to the end of the matched string. If the search string is not found between the current point and the end of the buffer, an error is signalled.

The search can be controlled by entering one of the following key sequences at any time during the search.

- **Ctrl+S**  
  If the search string is empty, repeats the last incremental search, otherwise repeats a forward search for the current search string.
If the search string cannot be found, starts the search from the beginning of the buffer (wrap-around search).

**Ctrl+R**  Changes to a backward (reverse) search.

**Delete**  Cancels the last character typed.

**Ctrl+Q**  Quotes the next character typed.

**Ctrl+W**  Adds the next word under the cursor to the search string.

**Meta+Ctrl+Y**  Adds the next form under the cursor to the search string.

**Ctrl+Y**  Adds the remainder of the line under the cursor to the search string.

**Meta+Y**  Adds the current kill string to the search string.

**Ctrl+C**  Add the editor window’s selected text to the search string.

**Esc**  If the search string is empty, invokes a non-incremental search, otherwise exits the search, leaving the current point at the last find.

**Ctrl+G**  Aborts the search, returning the current point to its original location.

If the search string cannot be found, cancels the last character typed (equivalent to **Delete**).

**Return**  Exits the search, leaving the current point at the last find.

---

**Reverse Incremental Search**  
*Editor Command*

Arguments: *string*

Key sequence: **Ctrl+R**  *string*
Searches backward, starting from the current point, for the search string that is input, beginning the search as soon as each character is provided. When a match is found for the search string, the current point is moved to the start of the matched string. If the search string is not found between the current point and the beginning of the buffer, an error is signalled.

The search can be controlled by entering one of the following key sequences at any time during the search.

- **Ctrl+R**  
  If the search string is empty, repeats the last incremental search, otherwise repeats a backward search for the current search string.  
  If the search string cannot be found, starts the search from the end of the buffer (wrap-around search).

- **Ctrl+S**  
  Changes to a forward search.

- **Delete**  
  Cancels the last character typed.

- **Esc**  
  If the search string is empty, invokes a non-incremental search, otherwise exits the search, leaving the current point at the last find.

- **Ctrl+G**  
  Aborts the search, returning the current point to its original location.  
  If the search string cannot be found, cancels the last character typed (equivalent to **Delete**).

- **Ctrl+Q**  
  Quotes the next character typed.

---

**Forward Search**

*Editor Command*

Arguments: *string*  
Key sequence: **Ctrl+S Esc** *string*

```
editor:forward-search-command p &optional string the-point
```

The default for *the-point* is the current point.
Searches forwards from the-point for string. When a match is found, the-point is moved to the end of the matched string. In contrast with Incremental Search, the search string must be terminated with a carriage return before any searching is done. If an empty string is provided, the last search is repeated.

**Backward Search**  
*Editor Command*

Arguments: *string*  
Key sequence: None  

```lisp
editor:reverse-search-command p &optional string the-point
```

The default for the-point is the current point.

Searches backwards from the-point for string. When a match is found, the-point is moved to the start of the matched string. In contrast with Reverse Incremental Search, the search string must be terminated with a carriage return before any searching is done. If an empty string is provided, the last search is repeated.

Reverse Search is a synonym for Backward Search.

**List Matching Lines**  
*Editor Command*

Arguments: *string*  
Key sequence: None  

```lisp
editor:list-matching-lines-command p &optional string
```

Lists all lines after the current point that contain *string*, in a Matches window.

A prefix argument causes the appropriate number of lines before and after each matching line to be listed also.

**Delete Matching Lines**  
*Editor Command*

Arguments: *string*  
Key sequence: None
Delete Non-Matching Lines

Arguments: string
Key sequence: None

Delete Non-Matching Lines  Editor Command

(editor:delete-non-matching-lines-command p &optional string)
Deletes all lines after the current point that do not match string.

Search All Buffers

Arguments: string
Key sequence: None

Search All Buffers  Editor Command

Searches all the buffers for string. If only one buffer contains string, it becomes the current one, with the cursor positioned at the start of the string. If more than one buffer contains the string, a popup window displays a list of those buffers. A buffer may then be selected from this list.

Directory Search

Arguments: directory string
Key sequence: None

Directory Search  Editor Command

Searches files in directory for string. The current working directory is offered as a default for directory.

By default only files with suffix .lisp, .lsp, .c or .h are searched. A non-nil prefix argument causes all files to be searched, except for those ending with one of the strings in the list system:*ignorable-file-suffices*.

Use the key sequence Meta+, to find subsequent definitions of the search string.
Search Files  
**Editor Command**

Arguments: search-string directory
Key sequence: Ctrl+X *

Searches for a string in a directory using a Search Files tool.

The command prompts for search-string and directory and then raises a Search Files tool. The configuration of the Search Files tool controls which files in the directory are searched. If the search string is not empty, it starts searching automatically, unless a prefix argument is given.

See the *LispWorks IDE User Guide* for a description of the Search Files tool.

Search Files Matching Patterns  
**Editor Command**

Arguments: search-string directory patterns
Key sequence: Ctrl+X &

Searches for a string in files under a directory with names matching given patterns, using a Search Files tool.

The command prompts for search-string, directory and patterns, and raises a Search Files tool in Roots and Patterns mode. If the search string is not empty, it starts searching automatically, unless a prefix argument is given.

patterns should be a comma-separated set of filename patterns delimited by braces. A pattern where the last component does not contain * is assumed to be a directory onto which the Search Files tool adds its own filename pattern. patterns defaults to {*.lisp,*.lsp,*.c,*.h}.

See the *LispWorks IDE User Guide* for a description of the Search Files tool.

System Search  
**Editor Command**

Arguments: system string
Key sequence: None

Searches the files of system for string.

Matches are shown in editor buffers consecutively. Use the key sequence Meta+, to find subsequent definitions of the search string.
3.21 Searching and replacing

Search System

Editor Command

Arguments: search-string system
Key sequence: None

Prompts for search-string and system and then raises a Search Files tool in System Search mode, which displays the search results and allows you to visit the files.

See the LispWorks IDE User Guide for a description of the Search Files tool.

default-search-kind

Editor Variable

Default value: :string-insensitive

Defines the default method of searching. By default, all searching (including regexp searching, and replacing commands) ignores case. If you want searching to be case-sensitive, the value of this variable should be set to :string-sensitive using Set Variable.

It is also possible to search a set of files programatically using the search-files function:

search-files

Function

Summary Search all the files in a list for a string.

Package editor

Signature search-files &key string files generator => nil

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>A string to search for (prompted if not given)</td>
</tr>
<tr>
<td>files</td>
<td>A list of pathnames of files to search</td>
</tr>
<tr>
<td>generator</td>
<td>A function to generate the files if none are given</td>
</tr>
</tbody>
</table>

Values search-files returns nil.
Description  search-files searches all the files in the list for a given string. If a match is found the file is loaded into a buffer with the cursor on the occurrence. Meta+-, makes the search continue until the next occurrence.

Example  CL-USER 91 > (editor:search-files:
/files '(".login" ".cshrc")
/string "alias")

3.21.2 Regular expression searching

A regular expression (regexp) allows the specification of the search string to include wild characters, repeated characters, ranges of characters, and alternatives. Strings which follow a specific pattern can be located, which makes regular expression searches very powerful.

The regular expression syntax used is similar to that of Emacs. In addition to ordinary characters, a regular expression can contain the following special characters to produce the search pattern:

. Matches any single character except a newline. For example, c.r matches any three character string starting with c and ending with r.

* Matches the previous regexp any number of times (including 0 times). For example, ca+r matches strings beginning with c and ending with r, with any number of a’s in-between.

An empty regexp followed by * matches an empty part of the input. By extension, ^* will match exactly what ^ matches.

+ Matches the previous regexp any number of times, but at least once. For example, ca+r matches strings beginning with c and ending with r, with at least one a in-between. An empty regexp followed by + matches an empty part of the input.
3.21 Searching and replacing

? Matches the previous regexp either 0 or 1 times. For example, ca?r matches either the string cr or car, and nothing else. An empty regexp followed by ? matches an empty part of the input.

^ Matches the next regexp as long as it is at the beginning of a line. For example, ^foo matches the string foo as long as it is at the beginning of a line.

$ Matches the previous regexp as long as it is at the end of a line. For example, foo$ matches the string foo as long as it is at the end of a line.

[ ] Contains a character set to be used for matching, where the other special characters mentioned do not apply. The empty string is automatically part of the character set. For example, [a,b] matches either a or b or the empty string. The regexp c[a-d]*r matches strings beginning with c and ending with r, with any number of a’s and d’s in-between.

The characters - and ^ have special meanings inside character sets. - defines a range and ^ defines a complement character set. For example, [a-d] matches any character in the range a to d inclusive. [^ab] matches any character except a or b.

\ Quotes the special characters. For example, \* matches the character * (that is, * has lost its special meaning).

\| Specifies an alternative. For example, ab\|cd matches either ab or cd.

\(, \) Provides a grouping construct. For example, ab\(cd\|ef\) matches either abcd or abef.
Regexp Forward Search

Editor Command

Arguments: string
Key sequence: Meta+Ctrl+S string

`editor:regexp-forward-search-command p &optional string the-point limit`

Performs a forward search for string using regular expressions. The search pattern must be terminated with a carriage return before any searching is done. If an empty string is provided, the last regexp search is repeated.

The argument `the-point` specifies the position from which the search is to start. The default is the current point. `limit` specifies a limiting point in the buffer for the search. The default is the end of the buffer.

Regexp Reverse Search

Editor Command

Arguments: string
Key sequence: Meta+Ctrl+R string

`editor:regexp-reverse-search-command p &optional string the-point limit`

Performs a backward search for string using regular expressions. The search pattern must be terminated with a carriage return before any searching is done. If an empty string is provided, the last regexp search is repeated.

The argument `the-point` specifies the position from which the search is to start. The default is one position before the current point. `limit` specifies a limiting point in the buffer for the search. The default is the current point.

Count Occurrences

Editor Command

Arguments: None
Default binding: None

`editor:count-occurrences-command p &optional regexp`

Counts the number of regular expression matches for the string `regexp` between the current point and the end of the buffer.

`Count Matches` is a synonym for `Count Occurrences`. 
3.21 Searching and replacing

3.21.3 Replacement

Replace String

*Editor Command*

Arguments: *target replacement*

Key sequence: None

editor:replace-string-command p &optional *target replacement*

Replaces all occurrences of *target* string by *replacement* string, starting from the current point.

Whenever *replacement* is substituted for *target*, case may be preserved, depending on the value of the editor variable *case-replace.*

Query Replace

*Editor Command*

Arguments: *target replacement*

Key sequence: *Meta+Shift+% target replacement*

editor:query-replace-command p &optional *target replacement*

Replaces occurrences of *target* string by *replacement* string, starting from the current point, but only after querying the user. Each time *target* is found, an action must be indicated from the keyboard.

Whenever *replacement* is substituted for *target*, case may be preserved, depending on the value of the editor variable *case-replace.*

The following key sequences are used to control *Query Replace*:

- **Space** or **y**
  - Replace *target* by *replacement* and move to the next occurrence of *target.*

- **Delete**
  - Skip *target* without replacing it and move to the next occurrence of *target.*

- **.**
  - Replace *target* by *replacement* and then exit.

- **t**
  - Replace all subsequent occurrences of *target* by *replacement* without prompting.
Ctrl+R

Enter recursive edit. This allows the current occurrence of target to be edited. When this editing is completed, Exit Recursive Edit should be invoked. The next instance of target is then found.

Esc

Quit from Query Replace with no further replacements.

Directory Query Replace

Editor Command

Arguments: directory target replacement

Key sequence: None

Replaces occurrences of target string by replacement string for each file with the suffix .lisp or .lsp in directory, but only after querying the user. The current working directory is offered as a default for directory. A non-nil prefix argument causes all files to be searched, except for those ending with one of the strings in the list system:*ignoreable-file-suffixes*. Each time target is found, an action must be indicated from the keyboard. For details of possible actions see Query Replace.

System Query Replace

Editor Command

Arguments: system target replacement

Key sequence: None

Replaces occurrences of target string by replacement string, for each file in system, but only after querying the user. Each time target is found, an action must be indicated from the keyboard. For details of possible actions see Query Replace.

case-replace

Editor Variable

Default value: t

If the value of this variable is t, Replace String and Query Replace try to preserve case when doing replacements. If its value is nil, the case of the replacement string is as defined by the user.
3.22 Comparison

**Replace Regexp**

*Editor Command*

Arguments: *target replacement*

Key sequence: None

`editor:replace-regexp-command p &optional target replacement`

Replaces all matches of *target* regular expression by *replacement* string, starting from the current point.

See “Regular expression searching” for a description of regular expressions.

**Query Replace Regexp**

*Editor Command*

Arguments: *target replacement*

Key sequence: None

`editor:query-replace-regexp-command p &optional target replacement`

Replaces matches of *target* regular expression by *replacement* string, starting from the current point, but only after querying the user. Each time *target* is matched, an action must be indicated from the keyboard.

See “Regular expression searching” for a description of regular expressions, and Query Replace for the keyboard gestures available.

### 3.22 Comparison

This section describes commands which compare files and/or buffers against each other.

**Diff**

*Editor Command*

Arguments: [*file1*] [*file2]*

Key sequence: None

Compares the current buffer with another file.

A prefix argument makes it compare any two files, prompting you for both filenames.
### Diff Ignoring Whitespace

*Editor Command*

- **Arguments:** `[file1] [file2]`
- **Key sequence:** None
- Compares the current buffer with another file, like `Diff` but ignoring whitespace.
- A prefix argument is interpreted in the same way as by `Diff`.

### 3.23 Registers

Locations and regions can be saved in *registers*. Each register has a name, and reference to a previously saved register is by means of its name. The name of a register, which consists of a single character, is case-insensitive.

#### Point to Register

*Editor Command*

- **Arguments:** `name`
- **Key sequence:** `Ctrl+X / name`
- Saves the location of the current point in a register called `name`, where `name` is a single character.
- *Save Position* is a synonym for *Point to Register*.

#### Jump to Register

*Editor Command*

- **Arguments:** `name`
- **Key sequence:** `Ctrl+X J name`
- Moves the current point to a location previously saved in the register called `name`.
- *Jump to Saved Position* and *Register to Point* are both synonyms for *Jump to Register*.

#### Kill Register

*Editor Command*

- **Arguments:** `name`
- **Key sequence:** None
Kills the register called name.

**List Registers**

*Editor Command*

Arguments: None
Key sequence: None
Lists all existing registers.

**Copy to Register**

*Editor Command*

Arguments: name
Key sequence: Ctrl+X X name
Saves the region between the mark and the current point to the register called name.

Put Register is a synonym for Copy to Register.

**Insert Register**

*Editor Command*

Arguments: name
Key sequence: Ctrl+X G name
Copies the region from the register called name to the current point.

Get Register is a synonym for Insert Register.

### 3.24 Modes

A buffer can be in two kinds of mode at once: *major* and *minor*. The following two sections give a description of each, along with details of some commands which alter the modes.

In most cases, the current buffer can be put in a certain mode using the mode name as an Editor Command.

#### 3.24.1 Major modes

The major modes govern how certain commands behave and how text is displayed. Major modes adapt a few editor commands so that their use is more
appropriate to the text being edited. Some movement commands are affected by
the major mode, as word, sentence, and paragraph delimiters vary with the
mode. Indentation commands are very much affected by the major mode See
‘Indentation’ on page 3-61.

Major modes available in the LispWorks editor are as follows:

- **Fundamental mode.** Commands behave in their most general manner,
default values being used throughout where appropriate.

- **Text mode.** Used for editing straight text and is automatically loaded if the
  file name ends in `.txt`, `.text` or `.tx`.

- **Lisp mode.** Used for editing Lisp programs and is automatically loaded if
  the file name ends in `.lisp`, `.lsp`, `.lispworks`, `.slisp`, `.l`, `.mcl` or `.cl`.

- **Shell mode.** Used for running interactive shells.

The major mode of most buffers may be altered explicitly by using the com-
mands described below.

By default, Lisp mode is the major mode whenever you edit a file with type
`lisp` (as with several other file types). If you have Lisp source code in files with
another file type `foo`, put a form like this in your `.lispworks` file, adding your
file extension to the default set:

```lisp
(editor:define-file-type-hook
  ("lispworks" "lisp" "slisp" "l" "lsp" "mcl" "cl" "foo")
  (buffer type)
  (declare (ignore type))
  (setf (editor:buffer-major-mode buffer) "Lisp"))
```

**Fundamental Mode**

*Editor Command*

Arguments: None

Key sequence: None

Puts the current buffer into Fundamental mode.

**Text Mode**

*Editor Command*

Arguments: None

Key sequence: None
3.24 Modes

Puts the current buffer into Text mode.

**Lisp Mode**

*Editor Command*

Arguments: None

Key sequence: None

Puts the current buffer into Lisp mode. Notice how syntax coloring is used for Lisp symbols. Also the balanced parentheses delimiting a Lisp form at or immediately preceding the cursor are highlighted, by default in green.

### 3.24.2 Minor modes

The minor modes determine whether or not certain actions take place. Buffers may be in any number of minor modes. No command details are given here as they are covered in other sections of the manuals.

Minor modes available in the LispWorks editor are as follows:

- **Overwrite mode.** Each character that is typed overwrites an existing character in the text—see “Overwriting” on page 60.

- **Auto Fill mode.** Lines are broken between words at the right hand margin automatically, so there is no need to type Return at the end of each line—see “Filling” on page 64.

- **Abbrev mode.** Allows abbreviation definitions to be expanded automatically—see “Abbreviations” on page 96.

- **Execute mode.** Used by the Listener to make history commands available (see the LispWorks IDE User Guide).

### 3.24.3 Default modes

**default-modes**

*Editor Variable*

Default value: ("Fundamental")

This editor variable contains the default list of modes for new buffers.
3.24.4 Defining modes

New modes can be defined using the defmode macro.

**defmode**

**Function**

**Summary**
Defines new editor modes.

**Package**
editor

**Signature**
defmode name &key setup-function syntax-table key-bindings no-redefine vars cleanup-function major-p transparent-p precedence => nil

**Arguments**
- **name**
  A string containing the name of the mode being defined.
- **setup-function**
  Name of function which sets up a buffer in this mode.
- **key-bindings**
  A quoted list of key-binding directions.
- **no-redefine**
  If t, the mode cannot be re-defined. The default value is nil.
- **vars**
  A quoted list of Editor variables and values.
- **aliases**
  A quoted list of synonyms for name.
- **cleanup-function**
  Called upon exit from a buffer in this mode.
- **major-p**
  If t, the mode is defined as major, otherwise minor. The default value is nil.

**Values**
defmode returns nil.

**Description**
This function defines an Editor mode called *name*. By default, any mode defined is a minor one—specification of major-mode status is made by supplying t to the major-p argument.

defmode is essentially for the purposes of mode specification—not all of the essential definitions required to establish a new mode are included.
Editor mode are made in a `defmode` call. In the example, below, other required calls are shown.

`key-bindings` can be defined by supplying a quoted list of bindings, where a binding is a list containing as a first element the (string) name of the Editor command being bound, and as the second, the key binding description (see Chapter 6, “Advanced Features”, for example key-bindings).

The state of Editor variables can be changed in the definition of a mode. These are supplied as a quoted list `vars` of dotted pairs, where the first element of the pair is the (symbol) name of the editor variable to be changed, and the second is the new value.

Both `setup-function` and `cleanup-function` are called with the mode and the buffer locked. They can modify the buffer itself, but they must not wait for anything that happens on another process, and they must not modify the mode (for example by setting a variable in the mode), and must not try to update the display.

**Example**

Let us define a minor mode, `Foo`. `Foo` has a set-up function, called `setup-foo-mode`. All files with suffix `.foo` invoke `Foo`-mode.

Here is the `defmode` form:

```lisp
(editor:defmode "Foo" :setup-function 'setup-foo-mode)
```

The next piece of code makes `.foo` files invoke `Foo`-mode:

```lisp
(editor:define-file-type-hook ("foo") (buffer type)
   (declare (ignore type))
   (setf (editor:buffer-minor-mode buffer "Foo") t))
```

The next form defines the set-up function:
(defun setup-foo-mode (buffer)
  (setf (editor:buffer-major-mode buffer) "Lisp")
  (let ((pathname (editor:buffer-pathname buffer)))
    (unless (and pathname
                  (probe-file pathname))
      (editor:insert-string
       (editor:buffer-point buffer)
       #.(format nil ";;; -*- mode :foo -*-~2%(in-package "CL-USER")~2%"))))

Now, any files loaded into the Editor with the suffix .foo
invoke the Foo minor mode.

3.25 Abbreviations

Abbreviations (abbrevs) can be defined by the user, such that if an abbreviation is
typed at the keyboard followed by a word terminating character (such as Space
or ,), the expansion is found and used to replace the abbreviation. Typing can
thereby be saved for frequently used words or sequences of characters.

There are two kinds of abbreviations: global abbreviations, which are expanded in
all major modes; and mode abbreviations, which are expanded only in defined
major modes.

Abbreviations (both global and mode) are only expanded automatically when
Abbrev mode (a minor mode) is on. The default is for abbrev mode to be off.

All abbreviations that are defined can be saved in a file and reloaded during
later editor sessions.

Abbrev Mode

Editor Command

Arguments: None
Key sequence: None

Switches abbrev mode on if it is currently off, and off if it is currently on.
Only when in abbrev mode are abbreviations automatically expanded.

Add Mode Word Abbrev

Editor Command

Arguments: abbrev
Key sequence: Ctrl+X Ctrl+A abbrev
Defines a mode abbreviation for the word before the current point.

A positive prefix argument defines an abbreviation for the appropriate number of words before the current point. A zero prefix argument defines an abbreviation for all the text in the region between the mark and the current point. A negative prefix argument deletes an abbreviation.

**Inverse Add Mode Word Abbrev**  
*Editor Command*

Arguments: expansion  
Key sequence: Ctrl+X Ctrl+H expansion

Defines the word before the current point as a mode abbreviation for expansion.

**Add Global Word Abbrev**  
*Editor Command*

Arguments: abbrev  
Key sequence: Ctrl+X + abbrev

Defines a global abbreviation for the word before the current point.

A positive prefix argument defines an abbreviation for the appropriate number of words before the current point. A zero prefix argument defines an abbreviation for all the text in the region between the mark and the current point. A negative prefix argument deletes an abbreviation.

**Inverse Add Global Word Abbrev**  
*Editor Command*

Arguments: expansion  
Key sequence: Ctrl+X - expansion

Defines the word before the current point as a global abbreviation for expansion.

**Make Word Abbrev**  
*Editor Command*

Arguments: abbrev expansion mode  
Key sequence: None
editor:make-word-abbrev-command p &optional abbrev expansion mode

Defines an abbreviation for expansion without reference to the current point. The default value for mode is global.

**Abbrev Expand Only**

*Editor Command*

- Arguments: None
- Key sequence: None

Expands the word before the current point into its abbreviation definition (if it has one). If the buffer is currently in abbrev mode then this is done automatically on meeting a word defining an abbreviation.

**Word Abbrev Prefix Point**

*Editor Command*

- Arguments: None
- Key sequence: Meta+’

Allows the prefix before the current point to be attached to the following abbreviation. For example, if the abbreviation valn is bound to valuation, typing re followed by Meta+’, followed by valn, results in the expansion revaluation.

**Unexpand Last Word**

*Editor Command*

- Arguments: None
- Key sequence: None

Undoes the last abbreviation expansion. If this command is typed twice in succession, the previous abbreviation is restored.

**Delete Mode Word Abbrev**

*Editor Command*

- Arguments: abbrev
- Key sequence: None

editor:delete-mode-word-abbrev-command p &optional abbrev mode
Deletes a mode abbreviation for the current mode. A prefix argument causes all abbreviations defined in the current mode to be deleted.
The argument \textit{mode} is the name of the mode for which the deletion is to be applied. The default is the current mode.

\textbf{Delete Global Word Abbrev} \hfill \textit{Editor Command}

Arguments: \textit{abbrev}
Key sequence: None

\texttt{editor:delete-global-word-abbrev-command \ p \ &optional \ abbrev}

Deletes a global abbreviation. A prefix argument causes all global abbreviations currently defined to be deleted.

\textbf{Delete All Word Abbrevs} \hfill \textit{Editor Command}

Arguments: None
Key sequence: None

Deletes all currently defined abbreviations, both global and mode.

\textbf{List Word Abbrevs} \hfill \textit{Editor Command}

Arguments: None
Key sequence: None

Displays a list of all the currently defined abbreviations in an Abbrev window.

\textbf{Word Abbrev Apropos} \hfill \textit{Editor Command}

Arguments: \textit{search-string}
Key sequence: None

\texttt{editor:word-abbrev-apropos-command \ p \ &optional \ search-string}

Displays a list of all the currently defined abbreviations which contain \textit{search-string} in their abbreviation definition or mode. The list is displayed in an Abbrev window.
Edit Word Abbrevs

Arguments: None
Key sequence: None

Allows recursive editing of currently defined abbreviations. The abbreviation definitions are displayed in an Edit Word Abbrevs buffer, from where they can be can be added to, modified, or removed. This buffer can then either be saved to an abbreviations file, or Define Word Abbrevs can be used to define any added or modified abbreviations in the buffer. When editing is complete, Exit Recursive Edit should be invoked.

Write Word Abbrev File

Arguments: filename
Key sequence: None

Saves the currently defined abbreviations to filename. If no file name is provided, the default file name defined by the editor variable abbrev-pathname-defaults is used.

Append to Word Abbrev File

Arguments: filename
Key sequence: None

Appends all abbreviations that have been defined or redefined since the last save to filename. If no file name is provided, the default file name defined by the editor variable abbrev-pathname-defaults is used.

abbrev-pathname-defaults

Default value: abbrev.defns

Defines the default file name for saving the abbreviations that have been defined in the current buffer.
3.26 Keyboard macros

**Read Word Abbrev File**  
*Editor Command*

- Arguments: `filename`
- Key sequence: None

`editor:read-word-abbrev-file-command p &optional filename`

Reads previously defined abbreviations from `filename`. The format of each abbreviation must be that used by Write Word Abbrev File and Insert Word Abbrevs.

**Insert Word Abbrevs**  
*Editor Command*

- Arguments: None
- Key sequence: None

Inserts into the current buffer, at the current point, a list of all currently defined abbreviations. This is similar to Write Word Abbrev File, except that the abbreviations are written into the current buffer rather than a file.

**Define Word Abbrevs**  
*Editor Command*

- Arguments: None
- Key sequence: None

Defines abbreviations from the definition list in the current buffer. The format of each abbreviation must be that used by Write Word Abbrev File and Insert Word Abbrevs.

3.26 Keyboard macros

Keyboard macros enable a sequence of commands to be turned into a single operation. For example, if it is found that a particular sequence of commands is to be repeated a large number of times, they can be turned into a keyboard macro, which may then be repeated the required number of times by using Prefix Arguments.

Note that keyboard macros are only available for use during the current editing session.
**Define Keyboard Macro**  
*Editor Command*

Arguments: None  
Key sequence: `Ctrl+X Shift+(`

Begins the definition of a new keyboard macro. All the commands that are subsequently invoked are executed and at the same time combined into the newly defined macro. Any text typed into the buffer is also included in the macro. The definition is ended with `End Keyboard Macro`, and the sequence of commands can then be repeated with `Last Keyboard Macro`.

**End Keyboard Macro**  
*Editor Command*

Arguments: None  
Key sequence: `Ctrl+X Shift+)`

Ends the definition of a keyboard macro.

**Last Keyboard Macro**  
*Editor Command*

Arguments: None  
Key sequence: `Ctrl+X E`

Executes the last keyboard macro defined. A prefix argument causes the macro to be executed the required number of times.

**Name Keyboard Macro**  
*Editor Command*

Arguments: `name`  
Key sequence: None

`editor:name-keyboard-macro-command p &optional name`

Makes the last defined keyboard macro into a command called `name` that can subsequently be invoked by means of `Extended Command`.

**Keyboard Macro Query**  
*Editor Command*

Arguments: `action`  
Key sequence: `Ctrl+X Q action`
During the execution of a keyboard macro, this command prompts for an action. It is therefore possible to control the execution of keyboard macros while they are running, to a small extent.

The following actions can be used to control the current macro execution.

- **Space**: Continue with this iteration of the keyboard macro and then proceed to the next.
- **Delete**: Skip over the remainder of this iteration of the keyboard macro and proceed to the next.
- **Escape**: Exit from this keyboard macro immediately.

### 3.27 Echo area operations

There are a range of editor commands which operate only on the Echo Area (that is, the buffer where the user types in commands).

Although in many cases the key bindings have a similar effect to the bindings used in ordinary buffers, this is just for the convenience of the user. In fact the commands that are invoked are different.

#### 3.27.1 Completing commands

Many of the commands used in the Editor are long, in the knowledge that the user can use completion commands in the Echo Area, and so rarely has to type a whole command name. Details of these completion commands are given below.

**Complete Input**

*Editor Command*

- **Arguments**: None
- **Key sequence**: `Tab`

Completes the text in the Echo Area as far as possible, thereby saving the user from having to type in the whole of a long file name or command.

Use `Tab Tab` to produce a popup list of all possible completions.
**Complete Field**  
*Editor Command*

Arguments: None  
Key sequence: `Space`

Completes the current part of the text in the Echo Area. So, for a command that involves two or more words, if **Complete Field** is used when part of the first word has been entered, an attempt is made to complete just that word.

**Confirm Parse**  
*Editor Command*

Arguments: None  
Key sequence: `Return`

Terminates an entry in the Echo Area. The Editor then tries to parse the entry. If `Return` is typed in the Echo Area when nothing is being parsed, or the entry is erroneous, an error is signalled.

**Help on Parse**  
*Editor Command*

Arguments: None  
Key sequences: `?`, `Help`, `F1`

Displays a popup list of all possible completions of the text in the echo area.

### 3.27.2 Repeating echo area commands

The Echo Area commands are recorded in a history ring so that they can be easily repeated. Details of these commands are given below.

**Previous Parse**  
*Editor Command*

Arguments: None  
Key sequence: `Meta+P`

Moves to the previous command in the Echo Area history ring. If the current input is not empty and the contents are different from what is on the
top of the ring, then this input is pushed onto the top of the ring before the new input is inserted.

Next Parse

Editor Command

Arguments: None
Key sequence: Meta+N

Moves to the next most recent command in the Echo Area history ring. If the current input is not empty and the contents are different from what is on the top of the ring, then this input is pushed onto the top of the ring before the new input is inserted.

3.27.3 Movement in the echo area

Echo Area Backward Character

Editor Command

Arguments: None
Key sequence: Ctrl+B

Moves the cursor back one position (without moving into the prompt).

Echo Area Backward Word

Editor Command

Arguments: None
Key sequence: Meta+B

Moves the cursor back one word (without moving into the prompt).

Beginning Of Parse

Editor Command

Arguments: None
Key sequence: Meta+<

Moves the cursor to the location immediately after the prompt in the Echo Area.
3 Command Reference

**Beginning Of Parse or Line**

*Editor Command*

Arguments: None
Key sequence: Ctrl+A

Moves the cursor to the location at the start of the current line in multi-line input, or to the location immediately after the prompt in the Echo Area.

### 3.27.4 Deleting and inserting text in the echo area

**Echo Area Delete Previous Character**

*Editor Command*

Arguments: None
Key sequence: Delete

Deletes the previous character entered in the Echo Area.

**Echo Area Kill Previous Word**

*Editor Command*

Arguments: None
Key sequence: Meta+Delete

Kills the previous word entered in the Echo Area.

**Kill Parse**

*Editor Command*

Arguments: None
Key sequence: Ctrl+C Ctrl+U

Kills the whole of the input so far entered in the Echo Area.

**Insert Parse Default**

*Editor Command*

Arguments: None
Key sequence: Ctrl+C Ctrl+P

Inserts the default value for the parse in progress at the location of the cursor. It is thereby possible to edit the default. Simply typing Return selects the default without any editing.
3.28 Editor variables

Return Default
Editor Command

Arguments: None
Key sequence: Ctrl+C Ctrl+R
Uses the default value for the parse in progress. This is the same as issuing the command Insert Parse Default and then pressing Return immediately.

Insert Selected Text
Editor Command

Arguments: None
Key sequence: Ctrl+C Ctrl+C
Inserts the editor window’s selected text in the echo area.

3.28 Editor variables

Editor variables are parameters which affect the way that certain commands operate. Descriptions of editor variables are provided alongside the relevant command details in this manual.

Show Variable
Editor Command

Arguments: variable
Key sequence: None
Indicates the value of variable.

Set Variable
Editor Command

Arguments: variable value
Key sequence: None
Allows the user to change the value of variable.

3.29 Recursive editing

Recursive editing occurs when you are allowed to edit text while an editor command is executing. The mode line of the recursively edited buffer is enclosed in
square brackets. For example, when using the command `Query Replace`, the `Ctrl+R` option can be used to edit the current instance of the target string (that is, enter a recursive edit). Details of commands used to exit a recursive edit are given below.

**Exit Recursive Edit**  
*Editor Command*

- **Argument:** None
- **Key sequence:** Meta+Ctrl+Z

Exits a level of recursive edit, returning to the original command. An error is signalled if not in a recursive edit.

**Abort Recursive Edit**  
*Editor Command*

- **Argument:** None
- **Key sequence:** Ctrl+]

Aborts a level of recursive edit, quitting the unfinished command immediately. An error is signalled if not in a recursive edit.

### 3.30 Key bindings

The commands for modifying key bindings that are described below are designed to be invoked explicitly during each session with the Editor. If the user wishes to create key bindings which are set up every session, the function `editor:bind-key` should be used—see “Customizing default key bindings” on page 164.

**Bind Key**  
*Editor Command*

- **Argument:** `command` `key-sequence` `bind-type`
- **Key sequence:** None

Binds `command` (full command names must be used) to `key-sequence`. After entering `command`, enter the keys of `key-sequence` and press `Return`.  
`bind-type` can be either buffer, global or mode. If a `bind-type` of buffer or mode is selected, the name of the buffer or mode required must then be
entered. When a bind-type of buffer is selected, the current buffer is offered as a default. The default value for bind-type is "Global".

Unless a bind type of global is selected, the scope of the new key binding is restricted as specified. Generally, most key bindings are global. Note that the Echo Area is defined as a mode, and some commands (especially those involving completion) are restricted to the Echo Area.

### Bind String to Key

*Editor Command*

**Argument:** `string key-sequence bind-type`

**Key sequence:** None

Make `key-sequence` insert `string`.

After entering `string`, enter the keys of `key-sequence` and press `Return`.

`bind-type` is interpreted as in `Bind Key`.

### Delete Key Binding

*Editor Command*

**Argument:** `key-sequence bind-type`

**Key sequence:** None

Removes a key binding, so that the key sequence no longer invokes any command. The argument `bind-type` can be either buffer, global or mode. If a `bind-type` of buffer or mode is selected, the name of the buffer or mode required must then be entered. The default value for `bind-type` is "Global".

It is necessary to enter the kind of binding, because a single key sequence may sometimes be bound differently in different buffers and modes.

### Illegal

*Editor Command*

**Argument:** None

**Key sequence:** None

Signals an editor error with the message "Illegal command in the current mode" accompanied by a beep. It is sometimes useful to bind key sequences to this command, to ensure the key sequence is not otherwise bound.
3 Command Reference

**Do Nothing**

*Editor Command*

Argument: None  
Key sequence: None  

Does nothing. This is therefore similar to `Illegal`, except that there is no beep and no error message.

### 3.31 Running shell commands from the editor

The editor allows both single shell commands to be executed and also provides a means of running a shell interactively.

**Shell Command**

*Editor Command*

Argument: `command`  
Key sequence: `Meta+! command`

Executes the single shell command `command`. The output from the command is displayed in a Shell Output buffer. A prefix argument causes the output from the shell command to be sent to the `*terminal-io*` stream rather than the Shell Output buffer.

**Run Command**

*Editor Command*

Argument: `command`  
Key sequence: None  

Executes the single shell command `command` in a Shell window. When the command terminates, the subprocess is closed down.

**Shell**

*Editor Command*

Argument: None  
Key sequence: None  

Opens a Shell window which allows the user to run a shell interactively. The major mode of the buffer is Shell mode, and the minor mode is Execute mode so the history key bindings available in the Listener can also be used in the Shell window.
Whenever the working directory is changed within the shell, the editor attempts to keep track of these changes and update the default directory of the Shell buffer. When a shell command is issued beginning with a string matching one of the editor variables `shell-cd-regexp`, `shell-pushd-regexp` or `shell-poptd-regexp`, the editor recognises this command as a change directory command and attempt to change the default directory of the Shell buffer accordingly. If you have your own aliases for any of the shell change directory commands, alter the value of the appropriate variable. For example, if the value of `shell-cd-regexp` is `"cd"` and the shell command `cd ~programs/lisp` is issued, the next time the editor command `Wfind File` is issued, the default directory offered is `~programs/lisp`. If you find that the editor has not recognised a change directory command then the editor command `cd` may be used to change the default directory of the buffer.

**CD**

*Editor Command*

- **Arguments:** directory
- **Key sequence:** None
- **Mode:** Shell

Changes the directory associated with the current buffer to *directory*. The current directory is offered as a default.

**shell-cd-regexp**

*Editor Variable*

- **Default value:** "cd"
- **Mode:** Shell

A regular expression that matches the shell command to change the current working directory.

**shell-pushd-regexp**

*Editor Variable*

- **Default value:** "pushd"
- **Mode:** Shell

A regular expression that matches the shell command to push the current working directory onto the directory stack.
shell-popd-regexp

*Editor Variable*

Default value: *popd*
Mode: Shell

A regular expression that matches the shell command to pop the current working directory from the directory stack.

prompt-regexp-string

*Editor Variable*

Default value: "^\[^#]$%>]*\[^#]$%>]*^"
Mode: Shell

The regexp used to find the prompt in a Shell window. This variable is also used in the Listener.

Interrupt Shell Subjob

*Editor Command*

Argument: None
Key sequence: Ctrl+C Ctrl+C
Mode: Shell

Sends an interrupt signal to the subjob currently being run by the shell. This is equivalent to issuing the shell command Ctrl+C.

*Note:* this command does not work on Microsoft Windows.

Stop Shell Subjob

*Editor Command*

Argument: None
Key sequence: Ctrl+C Ctrl+Z
Mode: Shell

Sends a stop signal to the subjob currently being run by the shell. This is equivalent to issuing the shell command Ctrl+Z.

*Note:* this command does not work on Microsoft Windows.
3.32 Buffers, windows and the mouse

3.32.1 Buffers and windows

You can transfer text between LispWorks Editor buffers and ordinary windows using the commands described below.

**Copy to Cut Buffer**  
*Editor Command*

Argument: None  
Key sequence: None  
Copies the current region to the Cut buffer. The contents of the buffer may then be pasted into a window using the standard method for pasting (in UNIX this is usually achieved by clicking the middle mouse button).

**Insert Cut Buffer**  
*Editor Command*

Argument: None  
Key sequence: None  
Inserts the contents of the Cut buffer at the current point. You can put text from a window into the Cut buffer using the standard method for cutting text (usually by holding the left mouse button while dragging the mouse).
3.32.2 Actions involving the mouse

The functions to which the mouse buttons are bound are not true Editor Commands. As such, the bindings cannot be changed. Details of mouse button actions are given below.

Note that marks may also be set by using editor key sequences—see “Marks and regions” on page 44—but also note that a region must be defined either by using the mouse or by using editor key sequences, as the region may become unset if a combination of the two is used. For example, using Ctrl+Space to set a mark and then using the mouse to go to the start of the required region unsets the mark.

**left-button**

Moves the current point to the position of the mouse pointer.

**shift-left-button**

In Emacs emulation, this moves the current point to the location of the mouse pointer and sets the mark to be the end of the new current form or comment line.

**control-shift-left-button**

Invokes the Editor Command **Save Region**, saving the region between the current point and the mark at the top of the kill ring. If the last command was **control-shift-left-button**, the Editor Command **Kill Region** is invoked instead. This allows one click to save the region, and two clicks to save and kill it.

**middle-button**

If your mouse has a middle button, it pastes the current selection at the location of the mouse pointer.

**right-button**

Brings up a context menu, from which a number of useful commands can be invoked. The options include **Cut**, **Copy**, and **Paste**.

**shift-right-button**

Inserts the form or comment line at the location of the mouse pointer at the current point.
3.33 Miscellaneous

**Report Bug**  
*Editor Command*

Argument: None  
Key sequence: None  
Opens a window containing the template for reporting bugs in LispWorks. This template can then be filled in and emailed to Lisp Support.

**break-on-editor-error**  
*Editor Variable*

Default value: `nil`.  
Specifies whether an `editor:editor-error` generates a Lisp `cerror`, or whether it just displays a message in the Echo Area.

**Report Manual Bug**  
*Editor Command*

Argument: None  
Key sequence: None  
Opens a window containing the template for reporting bugs in the Lisp-Works documentation. This template can then be filled in and emailed to Lisp Support.

**Room**  
*Editor Command*

Argument: None  
Key sequence: None  
Displays information on the current status of the memory allocation for the host computer.

**Build Application**  
*Editor Command*

Argument: None  
Key sequence: None
Invokes the Application Builder in the LispWorks IDE and does a build. By default, it uses the current buffer as the build script. If given a prefix argument it prompts for a file to use as the build script.
There are a whole set of editor commands designed to facilitate editing of Lisp programs. These commands are designed to understand the syntax of the Lisp language and therefore allow movement over Lisp constructs, indentation of code, operations on parentheses and definition searching. Lisp code can also be evaluated and compiled directly from the editor.

To use some of these commands the current buffer should be in Lisp mode. For more information about editor modes, see “Modes” on page 91.

Commands are grouped according to functionality as follows:

- “Functions and definitions”
- “Forms”
- “Lists”
- “Comments”
- “Parentheses”
- “Documentation”
- “Evaluation and compilation”
- “Breakpoints”
- “Removing definitions”
4 Editing Lisp Programs

4.1 Automatic entry into lisp mode

Some source files begin with a line of this form

```lisp
;; -*- Mode: Common-Lisp; Author: m.mouse -*-
```
or this:

```lisp
;; -*- Mode: Lisp; Author: m.mouse -*-
```

A buffer is automatically set to be in Lisp mode when such a file is displayed. Alternatively, if you have files of Common Lisp code with extension other than .lisp, add the following code to your .lispworks file, substituting the extensions shown for your own. This ensures that Lisp mode is the major mode whenever a file with one of these extensions is viewed in the editor:

```lisp
(editor:define-file-type-hook
 ("lispworks" "lisp" "slisp" "el" "lsp" "mcl" "cl")
(buffer type)
 (declare (ignore type))
 (setf (editor:buffer-major-mode buffer) "Lisp"))
```

Another way to make a Lisp mode buffer is the command New Buffer, and you can put an existing buffer into Lisp mode via the command Lisp Mode.

4.2 Syntax coloring

When in Lisp mode, the LispWorks editor provides automatic Lisp syntax coloring and parenthesis matching to assist the editing of Lisp programs.

You can ensure a buffer is in Lisp mode as described in “Automatic entry into lisp mode”.

To modify the colors used in Lisp mode syntax coloring, use Preferences... > Environment > Styles > Colors And Attributes as described in the LispWorks IDE User Guide. Adjust the settings for the styles whose names begin with "Lisp".
4.3 Functions and definitions

4.3.1 Movement, marking and specifying indentation

**Beginning of Defun**

Arguments: None  
Key sequence: Meta+Ctrl+A  
Moves the current point to the beginning of the current top-level form. A positive prefix argument \( p \) causes the point to be moved to the beginning of the form \( p \) forms back in the buffer.

**End of Defun**

Arguments: None  
Key sequence: Meta+Ctrl+E  
Moves the current point to the end of the current top-level form. A positive prefix argument \( p \) causes the point to be moved to the end of the form \( p \) forms forward in the buffer.

**Mark Defun**

Arguments: None  
Key sequence: Meta+Ctrl+H  
Puts the mark at the end of the current top-level form and the current point at the beginning of the form. The definition thereby becomes the current region. If the current point is initially located between two top-level forms, then the mark and current point are placed around the previous top-level form.

**Defindent**

Arguments: no-of-args  
Key sequence: None
4 Editing Lisp Programs

Defines the number of arguments of the operator to be specially indented if they fall on a new line. The indent is defined for the operator name, for example `defun`.

`Defindent` affects the special argument indentation for all forms with that operator which you subsequently indent.

4.3.2 Definition searching

Definition searching involves taking a name for a function (or a macro, variable, editor command, and so on), and finding the actual definition of that function. This is particularly useful in large systems, where code may exist in a large number of source files.

Function definitions are found by using information provided either by LispWorks source recording or by a Tags file. If source records or Tags information have not been made available to LispWorks, then the following commands do not work. To make the information available to LispWorks, set the variable `dspec:*active-finders*` appropriately. See the LispWorks User Guide and Reference Manual for details.

Source records are created if the variable `*record-source-files*` is true when definitions are compiled, evaluated or loaded. See the LispWorks User Guide and Reference Manual for details.

Tag information is set up by the editor itself, and can be saved to a file for future use. For each file in a defined system, the tag file contains a relevant file name entry, followed by names and positions of each defining form in that file. Before tag searching can take place, there must exist a buffer containing the required tag information. You can specify a previously saved tag file as the current tag buffer, or you can create a new one using `Create Tags Buffer`. GNU Emacs tag files are fully compatible with LispWorks editor tag files.

Find Source

Editor Command

Argument: name
Key sequence: Meta+. name
Tries to find the source code for \textit{name}. The symbol under the current point is offered as a default value for \textit{name}. A prefix argument automatically causes this default value to be used.

If the source code for \textit{name} is found, the file in which it is contained is displayed in a buffer. When there is more than one definition for \textit{name}, \texttt{Find Source} finds the first definition, and \texttt{Meta+, (Continue Tags Search)} finds subsequent definitions.

\texttt{Find Source} searches for definitions according to the value of \texttt{dspec:*active-finders*}. You can control which source record information is searched, and the order in which these are searched, by setting this variable appropriately. See the \textit{LispWorks User Guide and Reference Manual} for details. There is an example setting for this variable in the configuration files supplied.

If \texttt{dspec:*active-finders*} contains the value \texttt{:tags}, \texttt{Find Source} prompts for the name of a tags file, and this is used for the current and subsequent searches.

The found source is displayed according to the value of \texttt{editor:*source-found-action*}. This depends on the buffer with the found definition being in Lisp mode. For information on how to ensure this for particular file types, see “Automatic entry into lisp mode” on page 118.

\textbf{Find Source for Dspec}  \hfill \textit{Editor Command}

\textbf{Argument: dspec}

\textbf{Key sequence: None}

This command is similar to \texttt{Find Source}, but takes a definition spec \texttt{dspec} instead of a name as its argument.

For example, given a generic function \texttt{foo} of one argument, with methods specializing on classes \texttt{bar} and \texttt{baz},

\texttt{Find Source for Dspec foo}

will find each method definition in turn (with the continuation via \texttt{Meta+}), whereas

\texttt{Find Source for Dspec (method foo (bar))}
finds only the definition of the method on \texttt{bar}.

\textbf{Find Command Definition} \hspace{1cm} \textit{Editor Command}

\begin{itemize}
  \item \textbf{Argument:} \textit{command}
  \item \textbf{Key sequence:} None
\end{itemize}

This command is similar to \texttt{Find Source}, but takes the name of an editor command, and tries to find its source code.

Except in the Personal Edition, you can use this command to find the definitions of the predefined editor commands. See the \textit{LispWorks User Guide and Reference Manual} chapter ”Customization of LispWorks” for details.

\textbf{Edit Editor Command} \hspace{1cm} \textit{Editor Command}

\begin{itemize}
  \item \textbf{Argument:} \textit{command}
  \item \textbf{Key sequence:} None
\end{itemize}

This is a synonym for \texttt{Find Command Definition}.

\textbf{View Source Search} \hspace{1cm} \textit{Editor Command}

\begin{itemize}
  \item \textbf{Argument:} \textit{function}
  \item \textbf{Key sequence:} None
\end{itemize}

Shows the results of the latest source search (initiated by \texttt{Find Source} or \texttt{Find Source for Dspec} or \texttt{Find Command Definition}) in the Find Definitions view of the Editor. See the chapter on the Editor tool in the \textit{LispWorks IDE User Guide} for more information about the Find Definitions view.

\textbf{List Definitions} \hspace{1cm} \textit{Editor Command}

\begin{itemize}
  \item \textbf{Argument:} \textit{name}
  \item \textbf{Key sequence:} None
\end{itemize}

List the definitions for \textit{name}. The symbol under the current point is offered as a default value for \textit{name}. A prefix argument automatically causes this default value to be used.
4.3 Functions and definitions

This command searches for definitions and shows the results in the Find Definitions view of the Editor tool instead of finding the first definition. It does not set up the Meta+, action.

See the chapter on the Editor tool in the *LispWorks IDE User Guide* for more information about the Find Definitions view.

**List Definitions For Dspec**

*Editor Command*

Argument: *dspec*

Key sequence: None

This command is similar to *List Definitions*, but takes a definition spec *dspec* instead of a name as its argument.

This command searches for definitions and shows the results in the Find Definitions view of the Editor tool instead of finding the first definition. This command does not set up the Meta+, action.

See the chapter on the Editor tool in the *LispWorks IDE User Guide* for more information about the Find Definitions view.

**Create Tags Buffer**

*Editor Command*

Argument: None

Key sequence: None

Creates a buffer containing Tag search information, for all the *.lisp* files in the current directory. If you want to use this information at a later date then save this buffer to a file (preferably a file called *TAGS* in the current directory).

The format of the information contained in this buffer is compatible with that of GNU Emacs tags files.

A prefix argument causes the user to be prompted for the name of a file containing a list of files, to be used for constructing the tags table.

**Find Tag**

*Editor Command*

Key sequence: Meta+?
Tries to find the source code for a name containing a partial or complete match a supplied string by examining the Tags information indicated by the value of `dspec:*active-finders*).

The text under the current point is offered as a default value for the string.

If the source code for a match is found, the file in which it is contained is displayed. When there is more than one definition, `Find Tag` finds the first definition, and `Meta+, (Continue Tags Search)` finds subsequent definitions.

The found source is displayed according to the value of `editor:*source-found-action*`.

If there is no tags information indicated by the value of `dspec:*active-finders*`, `Find Tag` prompts for the name of a tags file. The default is a file called `TAGS` in the current directory. If there is no such file, you can create one using `Create Tags Buffer`. If you want to search a different directory, specify the name of a tags file in that directory.

See the chapter on the `DSPEC` package in the `LispWorks User Guide and Reference Manual` for information on how to use the `dspec:*active-finders*` variable to control how this command operates. There is an example setting for this variable in the configuration files supplied.

See also `Find Source`, `Find Source for Dspec` and `Create Tags Buffer`.

**Tags Search**

*Editor Command*

Key sequence: None

Exhaustively searches each file mentioned in the Tags files indicated by the value of `dspec:*active-finders*` for a supplied string `string`. Note that this does not merely search for definitions, but for any occurrence of the string.

If `string` is found, it is displayed in a buffer containing the relevant file. When there is more than one definition, `Tags Search` finds the first definition, and `Meta+, (Continue Tags Search)` finds subsequent definitions.

If there is no Tags file on `dspec:*active-finders*`, `Tags Search` prompts for the name of a tags file. The default is a file called `TAGS` in the current directory.
directory. If there is no such file, you can create one using Create Tags Buffer. If you want to search a different directory, specify the name of a tags file in that directory.

**Continue Tags Search**

*Editor Command*

- **Argument:** None
- **Key sequence:** Meta+,

Searches for the next match in the current search. This command is only applicable if issued immediately after a Find Source, Find Source for Dspec, Find Command Definition, Edit Callers, Edit Callees, Find Tag or Tags Search command.

**Tags Query Replace**

*Editor Command*

- **Key sequence:** None

Allows you to replace occurrences of a supplied string *target* by a second supplied string *replacement* in each Tags file indicated by the value of *dspec:*active-finders*.

Each time *target* is found, an action must be specified from the keyboard. For details of the possible actions see Query Replace.

If there is no Tags file indicated by *dspec:*active-finders*, Tags Query Replace prompts for the name of a tags file. The default is a file called TAGS in the current directory. If there is no such file, you can create one using Create Tags Buffer.

**Visit Tags File**

*Editor Command*

- **Key sequence:** None

Prompts for a Tags file *file* and makes the source finding commands use it. This is done by modifying, if necessary, the value of *dspec:*active-finders*.

If *file* is already in *dspec:*active-finders*, this command does nothing.
If there are other Tags files indicated then Visit Tags File prompts for whether to add simply add file as the last element of dspec:*active-finders*, or to save the current value of dspec:*active-finders* and start a new list of active finders, setting dspec:*active-finders* to the new value (:internal file). In this case, the previous active finders list can be restored by the command Rotate Active Finders.

If the value :tags appears on the list dspec:*active-finders* then file replaces this value in the list.

If there is no tags information indicated then Visit Tags File simply adds file as the last element of dspec:*active-finders*.

Rotate Active Finders

Editor Command

Key sequence: Meta+Ctrl+.

Rotates the active finders history, activating the least recent one. This modifies the value of dspec:*active-finders*.

The active finders history can have length greater than 1 if Visit Tags File started a new list of active finders, or if a buffer associated with a TAGS file on dspec:*active-finders* was killed.

Visit Other Tags File is a synonym for Rotate Active Finders.

4.3.3 Tracing functions

The commands described in this section use the Common Lisp trace facility. Note that you can switch tracing on and off using dspec:tracing-enabled-p - see the LispWorks User Guide and Reference Manual for details of this.

Trace Function

Editor Command

Argument: function
Key sequence: None

This command traces function. The symbol under the current point is offered as a default value for function. A prefix argument automatically causes this default value to be used.
4.3 Functions and definitions

**Trace Function Inside Definition**  
*Editor Command*

Argument: *function*  
Key sequence: None

This command is like `Trace Function`, except that *function* is only traced within the definition that contains the cursor.

**Untrace Function**  
*Editor Command*

Argument: *function*  
Key sequence: None

This command untraces *function*. The symbol under the current point is offered as a default value for *function*. A prefix argument automatically causes this default value to be used.

**Trace Definition**  
*Editor Command*

Argument: None  
Key sequence: None

This command traces the function defined by the current top-level form.

**Trace Definition Inside Definition**  
*Editor Command*

Argument: None  
Key sequence: None

This command is like `Trace Definition`, except that with a non-nil prefix argument, prompts for a symbol to trace. Also, it prompts for a symbol naming a second function, and traces the first only inside this.

**Untrace Definition**  
*Editor Command*

Argument: None  
Key sequence: None

This command untraces the function defined by the current top-level form.
4 Editing Lisp Programs

**Break Function**  
*Editor Command*

Argument: *function*  
Key sequence: None

This command is like **Trace Function** but the trace is with :break t so that when *function* is entered, the debugger is entered.

**Break Function on Exit**  
*Editor Command*

Argument: *function*  
Key sequence: None

This command is like **Trace Function** but the trace is with :break-on-exit t so that when a called to *function* exits, the debugger is entered.

**Break Definition**  
*Editor Command*

Argument: None  
Key sequence: None

Like **Trace Definition** but the definition is traced with :break t.

**Break Definition on Exit**  
*Editor Command*

Argument: None  
Key sequence: None

Like **Trace Definition** but the definition is traced with :break-on-exit t.

### 4.3.4 Function callers and callees

The commands described in this section, require that LispWorks is producing cross-referencing information. This information is produced by turning source debugging on while compiling and loading the relevant definitions (see toggle-source-debugging in the *LispWorks User Guide and Reference Manual*).
4.3 Functions and definitions

**List Callers**

*Editor Command*

Argument: *dspec*

Key sequence: None

Produces a Function Call Browser window showing those functions that call the definition named by *dspec*. The name of the current top-level definition is offered as a default value for *dspec*. A prefix argument automatically causes this default value to be used.


**List Callees**

*Editor Command*

Argument: *dspec*

Key sequence: None

Produces a Function Call Browser window showing those functions that are called by the definition named by *dspec*. The name of the current top-level definition is offered as a default value for *dspec*. A prefix argument automatically causes this default value to be used.


**Show Paths To**

*Editor Command*

Argument: *dspec*

Key sequence: None

Produces a Function Call Browser window showing the callers of the definition named by *dspec*. The name of the current top-level definition is offered as a default value for *dspec*. A prefix argument automatically causes this default value to be used.

Show Paths From

Editor Command

Argument: \textit{dspec}
Key sequence: None

Produces a Function Call Browser window showing the function calls from the definition named by \textit{dspec}. The name of the current top-level definition is offered as a default value for \textit{dspec}. A prefix argument automatically causes this default value to be used.


Edit Callers

Editor Command

Argument: \textit{function}
Key sequence: None

Produces an Editor window showing the latest definition found for a function that calls \textit{function}. The name of the current top-level definition is offered as a default value for \textit{function}. A prefix argument automatically causes this default value to be used. The latest definitions of each of the other functions that call \textit{function} are available via the \texttt{Continue Tags Search} command.

Edit Callees

Editor Command

Argument: \textit{function}
Key sequence: None

Produces an Editor window showing the latest definition found for a function called by \textit{function}. The name of the current top-level definition is offered as a default value for \textit{function}. A prefix argument automatically causes this default value to be used. The latest definitions of each of the other functions that are called by \textit{function} are available via the \texttt{Continue Tags Search} command.
4.3 Functions and definitions

4.3.5 Indentation and Completion

Indent Selection or Complete Symbol

Editor Command

Argument: None
Key sequence: Tab
Mode: Lisp

Does Lisp indentation if there is a visible region. Otherwise, it attempts to indent the current line. If the current line is already indented correctly then it attempts to complete the symbol before the current point. See Complete Symbol for more details.

The prefix argument, if supplied, is interpreted as if by Indent Selection or Complete Symbol.

Indent or Complete Symbol

Editor Command

Argument: None
Key sequence: None

Attempts to indent the current line. If the current line is already indented correctly then it attempts to complete the symbol before the current point. See Complete Symbol for more details.

The prefix argument, if supplied, is interpreted as if by Indent or Complete Symbol.

Complete Symbol

Editor Command

Argument: predicate
Key sequence: Meta+Ctrl+I

Attempts to complete the symbol before the current point. If the string to be completed is not unique, a list of possible completions is displayed.

If the Use in-place completion preference is selected then the completions are displayed in a window which allows most keyboard gestures to be processed as ordinary editor input. This allows speedy reduction of the
number of possible completions, while you can select the desired completion with Return, Up and Down.

If predicate is non-nil then only symbols which are bound or fbound are offered amongst the possible completions.

**Abbreviated Complete Symbol**

*Editor Command*

Argument: predicate

Key sequence: Meta+I

Attempts to complete the symbol abbreviation before the current point. If the string to be completed is not unique, a list of possible completions is displayed.

A symbol abbreviation is a sequence of words (sequences of alphanumeric characters) separated by connectors (sequences of non-alphanumeric, non-whitespace characters). Each word (connector) is a prefix of the corresponding word (connector) in the expansions. Thus if you complete the symbol abbreviation w-o then **with-open-file** and **with-open-stream** are amongst the completions offered, assuming the COMMON-LISP package is visible.

If the **Use in-place completion** preference is selected then the completions are displayed in a window which allows most keyboard gestures to be processed as ordinary editor input. This allows speedy reduction of the number of possible completions, while you can select the desired completion with Return, Up and Down.

If predicate is non-nil then only symbols which are bound or fbound are offered amongst the possible completions.

### 4.3.6 Miscellaneous

**Buffer Changed Definitions**

*Editor Command*

Argument: None

Key sequence: None
4.3 Functions and definitions

Calculates which definitions that have been changed in the current buffer during the current LispWorks session, and displays these in the Changed Definitions tab of the Editor tool.

By default the reference point against which changes are calculated is the time when the file was last read into the buffer. A prefix argument equal to the value of the editor variable `prefix-argument-default` means the reference point is the last evaluation. A prefix argument of 1 means the reference point is the time the buffer was last saved to file.

Note: the most convenient to use this command is via the Editor tool. Switch it to the Changed Definitions tab, where you can specify the reference point for calculating the changes.

**Function Arglist**

*Editor Command*

Argument: `function`
Key sequence: `Meta+=function`

Prints the arguments expected by `function` in the Echo Area. The symbol under the current point is offered as a default value for `function`. A prefix argument automatically causes this default value to be used.

Note: example code showing how to use this command to display argument lists automatically is supplied with LispWorks, in the file `examples/editor/commands/space-show-arglist.lisp`

**Function Arglist Display**

*Editor Command*

Argument: None
Key sequence: `Ctrl+`

Shows or hides information about the operator in the current form. The command controls display of a special window (displayer) on top of the editor. The displayer shows the operator and its arguments, and tries to highlight the current argument (that is, the argument at the cursor position). If it does not recognize the operator of the current form, it tries the surrounding form, and if that fails it tries a third level of surrounding form.
Additionally, while the displayer is visible:

- **Ctrl+/** Controls whether the documentation string of the operator is also shown.
- **Ctrl++** Moves the displayer up.
- **Ctrl+-** Moves the displayer down.

You can dismiss the displayer by invoking the command again, or by entering **Ctrl+G**. On Cocoa and Windows it is dismissed automatically when the underlying pane loses the focus.

In the LispWorks IDE you can change the style of the highlighting by **Preferences... > Environment > Styles > Colors and Attributes > Arglist Highlight**.

### Function Argument List

**Editor Command**

Argument: *function*

Key sequence: **Ctrl+Shift+A function**

This command is similar to **Function Arglist**, except that the symbol at the head of the current form is offered as a default value for *function*, unless that symbol is a member of the list `editor:*find-likely-function-ignores*` in which case the second symbol in the form is offered as the default. A prefix argument automatically causes this default value to be used.

### Describe Class

**Editor Command**

Argument: *class*

Key sequence: None

Displays a description of the class named by *class* in a Class Browser tool. The symbol under the current point is offered as a default value for *class*. A prefix argument automatically causes this default value to be used.
Describe Generic Function

Editor Command

Argument: function
Key sequence: None

Displays a description of function in a Generic Function Browser tool. The symbol under the current point is offered as a default value for function. A prefix argument automatically causes this default value to be used.

Describe Method Call

Editor Command

Argument: None
Key sequence: None

Displays a Generic Function Browser tool, with a specific method combination shown.

When invoked with a prefix argument p while the cursor is in a defmethod form, it uses the generic function and specializers of the method to choose the method combination.

Otherwise, it prompts for the generic function name and the list of specializers, which can be class names or lists of the form (eql object) where object is not evaluated.

Describe System

Editor Command

Argument: system
Key sequence: None

Displays a description of the defsystem-defined system named by system. The symbol under the current point is offered as a default value for system. A prefix argument automatically causes this default value to be used.
4.4 Forms

4.4.1 Movement, marking and indentation

Forward Form

Argument: None
Key sequence: Meta+Ctrl+F

Moves the current point to the end of the next form. A positive prefix argument causes the point to be moved the required number of forms forwards.

Backward Form

Argument: None
Key sequence: Meta+Ctrl+B

Moves the current point to the beginning of the previous form. A positive prefix argument causes the point to be moved the required number of forms backwards.

Mark Form

Argument: None
Key sequence: Meta+Ctrl+@

Puts the mark at the end of the current form. The current region is that area from the current point to the end of form. A positive prefix argument puts the mark at the end of the relevant form.

Indent Form

Argument: None
Key sequence: Meta+Ctrl+Q

If the current point is located at the beginning of a form, the whole form is indented in a manner that reflects the structure of the form. This command
can therefore be used to format a whole definition so that the structure of the definition is apparent.

See `editor:*indent-with-tabs*` for control over the insertion of `\Tab` characters by this and other indentation commands.

### 4.4.2 Killing forms

**Forward Kill Form**

*Editor Command*

- **Argument:** None
- **Key sequence:** `Meta+Ctrl+K`

Kills the text from the current point up to the end of the current form. A positive prefix argument causes the relevant number of forms to be killed forwards. A negative prefix argument causes the relevant number of forms to be killed backwards.

**Backward Kill Form**

*Editor Command*

- **Argument:** None
- **Key sequence:** `Meta+Ctrl+Backspace`

Kills the text from the current point up to the start of the current form. A positive prefix argument causes the relevant number of forms to be killed backwards. A negative prefix argument causes the relevant number of forms to be killed forwards.

**Kill Backward Up List**

*Editor Command*

- **Argument:** None
- **Key sequence:** None

Kills the form surrounding the current form. The cursor must be on the opening bracket of the current form. The entire affected area is pushed onto the kill-ring. A prefix argument causes the relevant number of surrounding lists to be removed.

For example, given the following code, with the cursor on the second open-bracket:
(print (do-some-work 1 2 3))

Kill Backward Up List would kill the outer form leaving this:

(do-some-work 1 2 3)

Also available through the function `editor:kill-backward-up-list-command`.

Extract List is a synonym for Kill Backward Up List.

### 4.4.3 Macro-expansion of forms

**Macroexpand Form**

- **Editor Command**
- Argument: None
- Key sequence: `Ctrl+Shift+M`
- Macro-expands the form after the current point. The output is sent to the Output window. A prefix argument causes the output to be displayed in the current buffer.

**Walk Form**

- **Editor Command**
- Argument: None
- Key sequence: `Meta+Shift+M`
- Produces a macroexpansion of the form after the current point. The output is sent to the Output window. A prefix argument causes the output to be displayed in the current buffer.

**Note:** Walk Form does not expand the Common Lisp macros `cond`, `prog`, `prog*` and `multiple-value-bind`, though it does expand their subforms.

### 4.4.4 Miscellaneous

**Transpose Forms**

- **Editor Command**
- Argument: None
- Key sequence: `Meta+Ctrl+T`
4.5 Lists

Transposes the forms immediately preceding and following the current point. A zero prefix argument causes the forms at the current point and the current mark to be transposed. A positive prefix argument causes the form at or preceding the current point to be transposed with the form the relevant number of forms forward. A negative prefix argument causes the form at or preceding the current point to be transposed with the form the relevant number of forms backward.

**Insert Double Quotes For Selection**  
*Editor Command*

Argument: None  
Key sequence: Meta+"

Inserts a pair of double-quotes around the selected text, if any. If there is no selected text and a prefix argument $p$ is supplied, insert them around the $p$ following (or preceding) forms. Otherwise insert them at the current point. The point is left on the character after the first double-quote.

4.5 Lists

4.5.1 Movement

**Forward List**  
*Editor Command*

Argument: None  
Key sequence: Meta+Ctrl+N

Moves the current point to the end of the current list. A positive prefix argument causes the point to be moved the required number of lists forwards.

**Backward List**  
*Editor Command*

Argument: None  
Key sequence: Meta+Ctrl+P
Moves the current point to the beginning of the current list. A positive prefix argument causes the point to be moved the required number of lists backwards.

**Forward Up List**  
*Editor Command*  
Argument: None  
Key sequence: None  
Moves the current point to the end of the current list by finding the first closing parenthesis that is not matched by an opening parenthesis after the current point.

**Backward Up List**  
*Editor Command*  
Argument: None  
Key sequence: Meta+Ctrl+U  
Moves the current point to the beginning of the current list by finding the first opening parenthesis that is not matched by a closing parenthesis before the current point.

**Down List**  
*Editor Command*  
Argument: None  
Key sequence: Meta+Ctrl+D  
Moves the current point to a location down one level in the current list structure. A positive prefix argument causes the current point to be moved down the required number of levels.

### 4.6 Comments

**Set Comment Column**  
*Editor Command*  
Argument: None  
Key sequence: Ctrl+X ;
4.6 Comments

Sets the comment column to the current column. A positive prefix argument causes the comment column to be set to the value of the prefix argument.

The value is held in the editor variable `comment-column`.

**Indent for Comment**

*Editor Command*

Argument: None

Key sequence: `Meta+;`

Creates a new comment or moves to the beginning of an existing comment, indenting it appropriately (see `Set Comment Column`).

If the current point is in a line already containing a comment, that comment is indented as appropriate, and the current point is moved to the beginning of the comment. An existing double semicolon comment is aligned as for a line of code. An existing triple semicolon comment or a comment starting in column 0, is not moved.

A prefix argument causes comments on the next relevant number of lines to be indented. The current point is moved down the relevant number of lines.

If characters not associated with the comment extend past the comment column, a space is added before starting the comment.

**Insert Multi Line Comment For Selection**

*Editor Command*

Argument: None

Key sequence: `Meta+#`

Inserts multi line comment syntax around the selected text, if any. If there is no selected text and a prefix argument \( p \) is supplied, inserts them around \( p \) following (or preceding) forms. Otherwise it inserts them at the current point. The point is left on the first character inside the comment.
4 Editing Lisp Programs

**Up Comment Line**  
*Editor Command*

Argument: None  
Key sequence: Meta+P  
Moves to the previous line and then performs an Indent for Comment.

**Down Comment Line**  
*Editor Command*

Argument: None  
Key sequence: Meta+N  
Moves to the next line and then performs an Indent for Comment.

**Indent New Comment Line**  
*Editor Command*

Argument: None  
Key sequence: Meta+J or Meta+Newline  
Ends the current comment and starts a new comment on the next line, using the indentation and number of comment start characters from the previous line’s comment. If Indent New Comment Line is performed when the current point is not in a comment line, it simply acts as a Return.

**Kill Comment**  
*Editor Command*

Argument: None  
Key sequence: Meta+Ctrl+;  
Kills the comment on the current line and moves the current point to the next line. If there is no comment on the current line, the point is simply moved onto the next line. A prefix argument causes the comments on the relevant number of lines to be killed and the current point to be moved appropriately.

The comment is identified by matching against the value of comment-start.
4.7 Parentheses

**comment-begin**  
*Editor Variable*

Default value: `*;*`  
Mode: Lisp  
When the value is a string, it is inserted to begin a comment by commands like `Indent for Comment` and `Indent New Comment Line`.

**comment-start**  
*Editor Variable*

Default value: `*;*`  
Mode: Lisp  
A string that begins a comment. When the value is a string, it is inserted to start a comment by commands like `Indent New Comment Line`, or used to identify a comment by commands like `Kill Comment`.

**comment-column**  
*Editor Variable*

Default value: 0  
Mode: Lisp  
Column to start comments in. Set by `Set Comment Column`.

**comment-end**  
*Editor Variable*

Default value: `nil`  
Mode: Lisp  
String that ends comments. The value `nil` indicates Newline termination. If the value is a string, it is inserted to end a comment by commands like `Indent New Comment Line`.

### 4.7 Parentheses

**Insert ()**  
*Editor Command*

Argument: None  
Key sequence: None
Inserts a pair of parentheses, positioning the current point after the opening parenthesis. A prefix argument \( p \) causes the parentheses to be placed around \( p \) following (or preceding) forms.

**Insert Parentheses For Selection**  
*Editor Command*

Argument: None  
Key sequence: Meta+({

Inserts a pair of parentheses around the selected text, if any. If there is no selected text and a prefix argument \( p \) is supplied, inserts them around \( p \) following (or preceding) forms. Otherwise it inserts them at the current point. The point is left on the character after the opening parenthesis.

**highlight-matching-parens**  
*Editor Variable*

Default value: \( t \)  
Mode: Lisp

When the value is true, matching parentheses are displayed in a different font when the cursor is directly to the right of the corresponding close parenthesis.

**Move Over )**  
*Editor Command*

Argument: None  
Key sequence: Meta+

Inserts a new line after the next closing parenthesis, moving the current point to the new line. Any indentation preceding the closing parenthesis is deleted, and the new line is indented.

**Lisp Insert )**  
*Editor Command*

Argument: None  
Key sequence: )  
Mode: Lisp
Inserts a closing parenthesis and highlights the matching opening parenthesis, thereby allowing the user to examine the extent of the parentheses.

**Find Unbalanced Parentheses**

*Editor Command*

Argument: None  
Key sequence: None  
Moves the point to the end of the last properly matched form, thereby allowing you to easily identify any parentheses in your code which are unbalanced.

*Find Mismatch* is a synonym for *Find Unbalanced Parentheses*.

**4.8 Documentation**

**Apropos**

*Editor Command*

Argument: `string`  
Key sequence: None  
Displays a Symbol Browser tool which lists symbols with symbol names matching `string`. The symbol name at the current point is offered as a default value for `string`.

By default `string` is matched against symbol names as a regular expression. A prefix argument causes a plain substring match to be used instead.

See “Regular expression searching” on page 84 for a description of regular expression matching. See the *LispWorks IDE User Guide* for a description of the Symbol Browser tool.

**Describe Symbol**

*Editor Command*

Argument: `symbol`  
Key sequence: None  
Displays a description (that is, value, property list, package, and so on) of `symbol` in a Help window. The symbol under the current point is offered as
a default value for string. A prefix argument automatically causes this
default value to be used.

**Function Documentation**

*Editor Command*

Arguments: None
Key sequence: Ctrl+Shift+D

```lisp
editor:function-documentation-command p```

Prompts for a symbol, which defaults to the symbol at the current point,
and displays the HTML documentation for that symbol if it is found in the
HTML manuals index pages.

On X11/Motif the prefix argument controls whether a new browser win-
dow is created. If the option **Reuse existing browser window** is selected in
the browser preferences, then the prefix argument causes the command to
create a new browser window. If **Reuse existing browser window** is dese-
lected, then the prefix argument causes the command to reuse an existing
browser window.

**Show Documentation**

*Editor Command*

Argument: *name*
Key sequence: Meta+Ctrl+Shift+A

Displays a Help window containing any documentation for the Lisp sym-
bol *name* that is present in the Lisp image. This includes function lambda
lists, and documentation strings accessible with `cl:documentation`, if any
such documentation exists.

**Show Documentation for Dspec**

*Editor Command*

Argument: *dspec*
Key sequence: None

Displays any documentation in the Lisp image for the dspec *dspec*, as
described for **Show Documentation**.
dspec is a symbol or list naming a definition, as described in the *LispWorks User Guide and Reference Manual*.

4.9 Evaluation and compilation

The commands described below allow the user to evaluate (interpret) or compile Lisp code that exists as text in a buffer. In some cases, the code may be used to modify the performance of the Editor itself.

4.9.1 General Commands

current-package

*Editor Variable*

Default value: nil

If non-nil, defines the value of the current package.

Set Buffer Package

*Editor Command*

Argument: package

Key sequence: None

Set the package to be used by Lisp evaluation and compilation while in this buffer. Not to be used in the Listener, which uses the value of *package* instead.

Set Buffer Output

*Editor Command*

Argument: stream

Key sequence: None

Sets the output stream that evaluation results in the current buffer are sent to.
4.9.2 Evaluation commands

Evaluate Defun

Argument: None
Key sequence: Meta+Ctrl+X

Evaluates the current top-level form. If the current point is between two forms, the previous form is evaluated.

If the form is a defvar form, then the command may first make the variable unbound, according to the value of evaluate-defvar-action, and hence assign the new value. This is useful because, whilst defvar does not reassign the value of a bound variable, when editing a program it is likely that you do want the new value.

evaluate-defvar-action

Default value: :reevaluate-and-warn

This affects the behavior of Evaluate Defun and Compile Defun when they are invoked on a defvar form. The allowed values are:

:evaluate-and-warn
Do not make the variable unbound before evaluating the form, and warn that it was not redefined.

:evaluate
Do not make the variable unbound before evaluating the form, but do not warn that it was not redefined.

:reevaluate-and-warn
Make the variable unbound before evaluating the form, and warn that it was therefore redefined.

:reevaluate
Make the variable unbound before evaluating the form, but do not warn that it was therefore redefined.
4.9 Evaluation and compilation

Reevaluate Defvar

Editor Command

Argument: None
Key sequence: None

Evaluates the current top-level form if it is a defvar. If the current point is between two forms, the previous form is evaluated. The form is treated as if the variable is not bound.

Re-evaluate Defvar is a synonym for Reevaluate Defvar.

Evaluate Expression

Editor Command

Argument: expression
Key sequence: Esc Esc expression
Key sequence: Meta+Esc expression

Evaluates expression. The expression to be evaluated is typed into the Echo Area and the result of the evaluation is displayed there also.

Evaluate Last Form

Editor Command

Argument: None
Key sequence: Ctrl+X Ctrl+E

Evaluates the Lisp form preceding the current point.

Without a prefix argument, prints the result in the Echo Area. With a non-nil prefix argument, inserts the result into the current buffer.

Evaluate Region

Editor Command

Argument: None
Key sequence: Ctrl+Shift+E

Evaluates the Lisp forms in the region between the current point and the mark.
4 Editing Lisp Programs

Evaluate Buffer

<table>
<thead>
<tr>
<th>Argument: None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key sequence: None</td>
</tr>
</tbody>
</table>

Evaluates the Lisp forms in the current buffer.

Load File

<table>
<thead>
<tr>
<th>Argument: file</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key sequence: None</td>
</tr>
</tbody>
</table>

Loads file into the current eval server, so that all Lisp forms in the file are evaluated.

Toggle Error Catch

<table>
<thead>
<tr>
<th>Argument: None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key sequence: None</td>
</tr>
</tbody>
</table>

Toggles error catching for expressions evaluated in the editor. By default, if there is an error in an expression evaluated in the editor, a Notifier window is opened which provides the user with a number of options, including debug, re-evaluation and aborting of the editor command. However, this behavior can be changed by using Toggle Error Catch, so that in the event of an error, the error message is printed in the Echo Area, and the user is given no restart or debug options.

Evaluate Buffer Changed Definitions

<table>
<thead>
<tr>
<th>Argument: None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key sequence: None</td>
</tr>
</tbody>
</table>

Evaluates definitions that have been changed in the current buffer during the current LispWorks session (use Buffer Changed Definitions on page 132 to see which definitions have changed). A prefix argument equal to the value of prefix-argument-default causes evaluation of definitions changed since last evaluated. A prefix argument of 1 causes evaluation of definitions changed since last saved.
4.9 Evaluation and compilation

**Evaluate Changed Definitions**

*Editor Command*

Argument: None  
Key sequence: None

Evaluates definitions in all Lisp buffers that have been changed during the current LispWorks session. The effect of prefixes is the same as for **Evaluate Buffer Changed Definitions**.

**Evaluate System Changed Definitions**

*Editor Command*

Argument: `system`  
Key sequence: None

Evaluates definitions that have been changed in `system` during the current LispWorks session.

### 4.9.3 Evaluation in Listener commands

**Evaluate Defun In Listener**

*Editor Command*

Argument: `editp`  
Key sequence: None

This command works rather like **Evaluate Defun** in that it evaluates the current top-level form and handles `defvar` forms usefully. However, instead of doing the evaluation in the Editor window, it copies the form into a Listener window as if you had entered it there.

Normally the evaluation is done immediately, but if a prefix argument is given, the text is inserted into the Listener for you to edit before pressing `Return` to evaluate it.

An `in-package` form is inserted before the form when necessary, so this will change the current package in the Listener.

**Evaluate Last Form In Listener**

*Editor Command*

Argument: `editp`  
Key sequence: None
This command works rather like `Evaluate Last Form` in that it evaluates the Lisp form preceding the current point. However, instead of doing the evaluation in the Editor window, it copies the form into a Listener window as if you had entered it there.

Normally the evaluation is done immediately, but if a prefix argument is given, the text is inserted into the Listener for you to edit before pressing `Return` to evaluate it.

An `in-package` form is inserted before the form when necessary, so this will change the current package in the Listener.

**Evaluate Region In Listener**

*Editor Command*

Argument: `editp`

Key sequence: None

This command works rather like `Evaluate Region` in that it evaluates the Lisp forms in the current region. However, instead of doing the evaluation in the Editor window, it copies the forms into a Listener window as if you had entered them there.

Normally the evaluation is done immediately, but if a prefix argument is given, the forms are inserted into the Listener for you to edit before pressing `Return` to evaluate them.

An `in-package` form is inserted before the forms when necessary, so this will change the current package in the Listener.

### 4.9.4 Compilation commands

**Compile Defun**

*Editor Command*

Argument: None

Key sequence: `Ctrl+Shift+C`

Compiles the current top-level form. If the current point is between two forms, the previous form is evaluated.

If the form is a `defvar` form, then the command may first make the variable unbound, according to the value of `evaluate-defvar-action`, and
4.9 Evaluation and compilation

hence assign the new value. This is useful because, whilst defvar does not reassign the value of a bound variable, when editing a program it is likely that you do want the new value.

**Compile Region**

*Editor Command*

- Argument: None
- Key sequence: Ctrl+Shift+R
- Compiles the Lisp forms in the region between the current point and the mark.

**Compile File**

*Editor Command*

- Argument: `file`
- Key sequence: None
- Compiles `file` unconditionally, with cl:compile-file.
- No checking is done on write dates for the source and binary files, to see if the file needs to be compiled. Also, no checking is done to see if there is a buffer for the file that should first be saved.

**Compile Buffer**

*Editor Command*

- Argument: None
- Key sequence: Ctrl+Shift+B
- Reads, compiles and then executes in turn each of the Lisp forms in the current buffer.

**Compile Buffer File**

*Editor Command*

- Argument: None
- Key sequence: None
- Compiles the source file in the current buffer as if by Compile File, but checks the file first. If its associated binary (fasl) file is older than the source file or does not exist then the file is compiled. If the binary file is up to date, the user is asked whether the file should be compiled anyway.
When `compile-buffer-file-confirm` is true, the user is always asked for confirmation, even when the date of the source file is later than that of the binary file.

A prefix argument causes the file to be compiled without checking the date or existence of the binary file.

**compile-buffer-file-confirm**  
*Editor Variable*

Default value: `t`  
Determines whether `Compile Buffer File` should prompt for a compilation to proceed. If the value is true, the user is always prompted for confirmation.

**Compile Buffer Changed Definitions**  
*Editor Command*

Argument: None  
Key sequence: None  
Compiles definitions that have been changed in the current buffer during the current LispWorks session (use `Buffer Changed Definitions` to see which definitions have changed). A prefix argument equal to the value of `prefix-argument-default` causes compilation of definitions changed since last compiled. A prefix argument of 1 causes compilation of definitions changed since last saved.

**Compile Changed Definitions**  
*Editor Command*

Argument: None  
Key sequence: None  
Compiles definitions in all Lisp buffers that have been changed during the current LispWorks session. The effect of prefixes is the same as for `Compile Buffer Changed Definitions`. 
Compile System

Argument: system
Key sequence: None
Compiles all files in the system system.
If ASDF is loaded and the LispWorks tools are configured to use it, then
this command works with ASDF systems as well as those defined by
lispworks:defsystem.

Compile System Changed Definitions

Argument: system
Key sequence: None
Compiles definitions that have been changed in system during the current
LispWorks session.

Disassemble Definition

Argument: definition
Key sequence: None
Outputs assembly code for definition to the Output window, compiling it
first if necessary. The name of the current top-level definition is offered as
a default value for definition.

Edit Recognized Source

Argument: None
Key sequence: Ctrl+X ,
Edit the source of the next compiler message, warning or error. It should
be used while viewing the Output window. Without a prefix argument, it
searches forwards in the Output window until it finds text which it
recognizes as a compiler message, warning or error, and then shows the
source code associated with that message. With a prefix argument, it
searches backwards.
4.10 Breakpoints

These commands operate on breakpoints, which are points in code where execution stops and the LispWorks IDE invokes the Stepper tool.


4.10.1 Setting and removing breakpoints

**Toggle Breakpoint**

*Editor Command*

Argument: None

Key sequence: None

If there is no breakpoint at the current point, sets a breakpoint there if possible. If there is a breakpoint at the current point, removes it.

4.10.2 Moving between breakpoints

**Next Breakpoint**

*Editor Command*

Argument: None

Key sequence: None

Moves the point to the next breakpoint in the current buffer. If given a numeric prefix argument $p$, it skips $p-1$ breakpoints.

**Previous Breakpoint**

*Editor Command*

Argument: None

Key sequence: None

Moves the point to the previous breakpoint in the current buffer. If given a numeric prefix argument $p$, it skips $p-1$ breakpoints.
4.11 Removing definitions

These commands allow the user to remove definitions from the running Lisp image. It uses Common Lisp functionality such as `fmakunbound`, `makunbound` and `remove-method` to undefine Lisp functions, variables, methods and so on.

Note: This does not mean deleting the source code.

4.11.1 Undefining one definition

**Undefine**

*Editor Command*

Argument: None
Key sequence: None

Without a prefix argument, this undefines the current top level definition. That is, the defining form around or preceding the current point.

With a non-nil prefix argument, this does not undefine the definition but instead inserts into the buffer a Lisp form which, if evaluated, would undefine the definition.

**Undefine Command**

*Editor Command*

Argument: None
Key sequence: None

Prompts for the name of an Editor command, and undefines that command.

4.11.2 Removing multiple definitions

**Undefine Buffer**

*Editor Command*

Argument: None
Key sequence: None

Undefines all the definitions in the current buffer.
4 Editing Lisp Programs

**Undefine Region**

*Editor Command*

- Argument: None
- Key sequence: None

Undefines the definitions in the current region.
By default the LispWorks Editor emulates GNU Emacs. This is often unusable for programmers familiar only with KDE/Gnome keys and behavior: for instance, a selection is not deleted on input, and most of the commonly used keys differ.

The LispWorks editor can be switched to emulate the KDE/Gnome model instead of Emacs.

When using KDE/Gnome emulation the main differences are:

- An alternate set of key bindings for the commonly-used commands.
- The abort gesture for the current editor command is Esc, not Ctrl+G.
- Inserted text replaces any currently selected text.
- The cursor is a vertical bar rather than a block.

### 5.1 Using KDE/Gnome editor emulation

To switch KDE/Gnome emulation on, use Preferences... > Environment > Emulation. See the section "Configuring the editor emulation" in the *LispWorks IDE User Guide* for details.
5.2 Key bindings

The key bindings for KDE/Gnome emulation are supplied in the LispWorks library file `config/msw-key-binds.lisp`. This file is loaded the first time that you use KDE/Gnome emulation, or on startup if your preference is stored.

5.2.1 Finding the keys

There are several ways to find the key for a given command, and the command on a given key:

- The files `msw-key-binds.lisp` and `selection-key-binds.lisp` show the default state, just like `key-binds.lisp` shows the Emacs bindings.
- The Editor command `Describe Bindings` shows all the current key bindings, including those specific to the buffer, the major mode and any minor modes that are in effect.
- The Editor command `Describe Key` reports the command on a given key.
- The Editor command `Where Is` reports the key for a given command.
- Use the `Help > Editing` menu.

5.2.2 Modifying the Key Bindings

As in Emacs emulation, the key sequences to which individual commands are bound can be changed, and key bindings can be set up for commands which are not, by default, bound to any key sequences.

Interactive means of modifying key bindings are described in “Key bindings” on page 108. Key bindings can also be defined programmatically via `editor:bind-key` forms similar to those in `msw-key-binds.lisp`.

However, note that you must use `editor:set-interrupt-keys` if you wish to alter the abort gesture.

5.2.3 Accessing Emacs keys

When KDE/Gnome emulation is on, Emacs keys are still available via the prefix `Ctrl+E`. For example, to invoke the command `WFind File`, enter:

```
Ctrl+E Ctrl+X Ctrl+F
```
5.2.4 The Alt modifier and editor bindings

In Microsoft Windows emulation on Microsoft Windows, keystrokes with the Alt modifier key are used by the system to activate the menu bar. Therefore these keystrokes, for example Alt+A and Alt+Ctrl+A are not available to the editor.

Windows accelerators always take precedence over editor key bindings, so in Emacs emulation the Alt modifier key only acts as Meta though keystrokes with Alt if there is no accelerator which matches.

On Cocoa, the preference for the Meta key affects the operation of menu accelerators (shortcuts). If Command is used as Meta, then it will not be available for use as an accelerator.

5.3 Replacing the current selection

When using KDE/Gnome emulation, Delete Selection Mode is active so that selected text is deleted when you type or paste text. Also, Delete deletes the current selection.

Note: Delete Selection Mode can also be used independently of KDE/Gnome emulation. See “Delete Selection” on page 56 for details.

5.4 Emulation in Applications

If you include the LispWorks editor (via capi:editor-pane or its subclasses) in an application, then by default your interfaces will use Microsoft Windows emulation on Windows, Mac OS editor emulation on Cocoa, and Emacs emulation on Unix and Linux.

To override this behavior in your interface classes, define a method on capi:interface-keys-style. See the LispWorks CAPI Reference Manual for details.

To override this behavior in your delivered application, use the delivery keyword :editor-style. See the LispWorks Delivery User Guide for details.
5 Emulation
The editor can be customized, both interactively and programmatically, to suit the users requirements.

The chapter “Command Reference” provides details of commands used to customize the editor for the duration of an editing session (see “Keyboard macros” on page 101, “Key bindings” on page 108, “Editor variables” on page 107). This chapter contains information on customizing the editor on a permanent basis.

There are a number of ways in which the editor may be customized:

- The key sequences to which individual commands are bound can be changed, and key bindings can be set up for commands which are not, by default, bound to any key sequences—see “Customizing default key bindings” on page 164.
- The indentation used for Lisp forms can be modified to suit the preferences of the user—see “Customizing Lisp indentation” on page 166.
- Additional editor commands can be created by combining existing commands and providing specified arguments for them—see “Programming the editor” on page 166.

Note that the default configuration files mentioned in this chapter were used when LispWorks was released. They are not read in when the system is run, so any modification to them will have no effect. If the user wishes to modify the
behavior of LispWorks in any of these areas, the modifying code should be
included in the .lispworks file, or an image containing the modifications
should be saved.

6.1 Customizing default key bindings

The key sequences to which individual commands are bound can be changed,
and key bindings can be set up for commands which are not, by default, bound
to any key sequences. Interactive means of modifying key bindings are
described in “Key bindings” on page 108.

This section describes the editor function bind-key, which is used to establish
bindings programmatically. If you want to alter your personal key bindings, put
the modifying code in your .lispworks file.

The default Emacs key bindings can be found in the file
config/key-binds.lisp in the LispWorks library directory. See “Key bindings”
for details of the key binds files used in other editor emulations.

editor:bind-key

Function

editor:bind-key name key &optional kind where

Binds the command name to the key sequence or combination key.

kind can take the value :global, :mode, or :buffer.

The default for kind is :global, which makes the binding apply in all buff-
ers and all modes, unless overridden by a mode-specific or buffer-specific
binding.

If where is not supplied, the binding is for the current emulation.
Otherwise where should be either :emacs or :pc, meaning that the binding
is for Emacs emulation or KDE/Gnome emulation respectively.

Note: before the editor starts, the current emulation is :emacs. Therefore
bind-key forms which do not specify where and which are evaluated
before the editor starts (for example, in your initialization file) will apply
to Emacs emulation only. Thus for example

(bind-key "Command" "Control-Right")
6.1 Customizing default key bindings

when evaluated in your initialization file will establish an Emacs emulation binding. The same form when evaluated after editor startup will establish a binding in the current emulation: Emacs or KDE/Gnome emulation.

It is best to specify the intended emulation:

(editor:bind-key "Command" "Control-Right" :global :pc)

If kind is :buffer the binding applies only to a buffer which should be specified by the value of where.

If kind is :mode the binding applies only to a mode which should be specified by where.

If this function is called interactively via the command Bind Key, you will be prompted as necessary for the kind of binding, the buffer or the mode. The binding is for the current emulation. Tab completion may be used at any stage.

The following examples, which are used to implement some existing key bindings, illustrate how key sequences can be specified using bind-key.

(editor:bind-key "Forward Character" \control-f)
(editor:bind-key "Forward Word" \meta-f)
(editor:bind-key "Save File" '('#\control-x #\control-s))
(editor:bind-key "Regexp Forward Search" \meta-control-s)
(editor:bind-key "Complete Field" \space :mode "Echo Area")
(editor:bind-key "Backward Character" "left")
(editor:bind-key "Forward Word" "("control-right")

editor:bind-string-to-key

Function

editor:bind-string-to-key string key &optional kind where

Binds the text string string to the keyboard shortcut key without the need to create a command explicitly. Using key inserts string in the current buffer. The kind and where arguments are as for editor:bind-key.

editor:set-interrupt-keys

Function

editor:set-interrupt-keys keys &optional input-style
The key that aborts the current editor command is handled specially by the editor. If you wish to change the default (from Ctrl+G for Emacs) then you must use this function rather than editor:bind-key. See the file config/msw-key-binds.lisp for an example.

### 6.2 Customizing Lisp indentation

The indentation used for Lisp forms can be modified to suit the preferences of the user.

The default indentations can be found in the file config/indents.lisp in the LispWorks library directory. If you want to alter your personal Lisp indentation, put the modifying code in your .lispworks file.

```lisp
(editor:setup-indent "form-name" no-of-args &optional standard special)
```

Modifies the indentation, in Lisp Mode, for the text following an instance of `form-name`. The arguments `no-of-args`, `standard` and `special` should all be integers. The first `no-of-args` forms following the `form-name` become indented `special` spaces if they are on a new line. All remaining forms within the scope of the `form-name` become indented `standard` spaces.

For example, the default indentation for `if` in Lisp code is established by:

```lisp
(editor:setup-indent "if" 2 2 4)
```

This determines that the first 2 forms after the `if` (that is, the `test` and the `then` clauses) get indented 4 spaces relative to the `if`, and any further forms (here, just an `else` clause) are indented by 2 spaces.

### 6.3 Programming the editor

The editor functions described in this section can be combined and provided with arguments to create new commands. Existing editor commands can also be used in the creation of new commands (see `editor:defcommand`).

Note that all the code found in this chapter is included in the directory examples/editor in the directory defined by the variable *lispworks-directory*.
6.3 Programming the editor

Note: code which modifies the contents of a `capi:editor-pane` (for example a displayed editor buffer) must be run only in the interface process of that pane.

The following sections describe editor functions that are not interactive editor commands.

### 6.3.1 Calling editor functions

All editor commands and some other editor functions expect to be called within a dynamic context that includes settings for the current buffer and current window. This happens automatically when using the editor interactively.

You can set up the context in a CAPI application by using the function `capi:call-editor` (see the LispWorks CAPI Reference Manual).

You can also use the following function to call editor commands and functions.

```lisp
editor:process-character
```

**Function**

`editor:process-character char window`

Processes `char` in a dynamic context where the current window is `window` and the current buffer is the buffer currently displayed in `window`.

The `char` can be one of the following:

- A string, naming an editor command to invoke.
- A list of the form `(function . args)`, which causes `function` to be called with `args`. The items in `args` are not evaluated.
- A function or symbol, which is called with `nil` as its argument (like a command function would be if there is no prefix argument).
- A character or `system:gesture-spec` object, which is treated as if it has been typed on the keyboard.

There is no return value. The processing may happen in another thread, so may not have competed before this function returns.
6.3.2 Defining commands

**defcommand**

*Macro*

**Summary**
Defines new editor commands.

**Package**
editor

**Signature**
defcommand name lambda-list command-doc function-doc &body forms
=> command-name

**Arguments**
- **name**  
The name of the new editor command. See Description for more details.
- **lambda-list**  
The lambda list of the new command, which must have at least one argument.
- **command-doc**  
A string which gives a detailed description of the command.
- **function-doc**  
A string which gives a brief description of the command.
- **forms**  
The Lisp code for the command.

**Values**
- **command-name**  
The symbol naming the new command. This symbol is interned in the current package.

**Description**
This macro defines a new editor command *name* that can be invoked in the editor by means of Extended Command. The macro takes the specification of the command as supplied, and creates a new Lisp function *command-name* from it.

**Note:** Every editor command has an associated Lisp function named *command-command*. For example:

```
editor:self-insert-command p &optional char
```

For every editor command documented in this manual, the associated *command-command* symbol is exported from the
editor package; and for every editor command created by user code, the associated command-command symbol is interned in the current package.

Existing editor commands can be used within the body of defcommand. To make use of an existing command, the command name should be hyphenated with a command suffix added. For example, the editor command Forward Character is referred to by forward-character-command. The syntax of a call to an existing command is the same as a call to a standard Lisp function. The first argument of all command definitions is the prefix argument, and this must therefore be included in any calls made to commands from defcommand, even when prefix arguments are ignored by the command. Some commands have additional optional arguments and details of these are provided in the command descriptions throughout this manual.

The name of the command must be a string, while the name of the associated function must be a symbol. There are two ways in which name can be supplied. Most simply, name is given as a string, and the string is taken to be the name of the editor command. The symbol the function needs as a name is computed from that string. Any spaces in the string are replaced with hyphens, and the quotes are discarded, but otherwise the symbol contains the same characters as the string.

If a specific function name, different to the one defcommand derives itself, is required, then this can be supplied explicitly, by passing a list as name. The first element of the list is the string used as the name of the command, while the last element is the symbol used to name the Lisp function.

The command-doc and function-doc variables may be empty strings if no documentation is available for them.

Example

The following code defines an editor command, Move Five, which moves the cursor forward in an editor buffer by five characters.
(editor:defcommand "Move Five" (p)
  "Moves the current point forward five characters."
  "Any prefix argument is ignored."
  "Moves five characters forward."
  (editor:forward-character-command 5))
=>
MOVE-FIVE-COMMAND

The first string gives the command’s name. This is the simple
form of definition, where no explicit name for the Lisp function
is given.

p is not used, and is there simply because the lambda-list must
have at least one element.

The second string is the command documentation, while the
third is the function documentation. After these, the Lisp code
defines what the command actually does.

Use Meta+X Move Five to invoke the command.

This command changes all the text in a writable buffer to be
uppercase:

(editor:defcommand "Uppercase Buffer" (p)
  "Uppercase the buffer contents***
  (declare (ignore p))
  (let* ((buffer (editor:current-buffer))
         (point (editor:buffer-point buffer))
         (start (editor:buffers-start buffer))
         (end (editor:buffers-end buffer)))
    (editor:set-current-mark start)
    (editor:move-point point end)
    (editor:uppercase-region-command nil)))

6.3.3 Buffers

Each buffer that you manipulate interactively using editor commands is an
object of type editor:buffer that can be used directly when programming the
editor. Buffers contain an arbitrary number of editor:point objects, which are
used when examining or modifying the text in a buffer (see “Points” on page
176).
6.3 Programming the editor

6.3.3.1 Buffer locking

Each buffer contains a lock that is used to prevent more than one thread from modifying the text, text properties or points within the buffer simultaneously. All of the exported editor functions (\texttt{editor:insert-string}, \texttt{editor:move-point} etc) claim this lock implicitly and are therefore atomic with respect to other such functions.

In situations where you want to make several changes as one atomic operation, use one of the macros \texttt{editor:with-buffer-locked} or \texttt{editor:with-point-locked} to lock the buffer for the duration of the operation. For example, if you want to delete the next character and replace it by a space:

\begin{verbatim}
(editor:with-buffer-locked ((editor:current-buffer))
 (editor:delete-next-character-command nil)
 (editor:insert-character (editor:current-point)
   \#\Space))
\end{verbatim}

In addition, you sometimes want to examine the text in a buffer without changing it, but ensure that no other thread can modify it in the meantime. This can be achieved by locking the buffer using \texttt{editor:with-buffer-locked} or \texttt{editor:with-point-locked} and passing the \texttt{for-modification} argument as \texttt{nil}. For example, if you are computing the beginning and end of some portion of the text in a buffer and then performing some operation on that text, you may want to lock the buffer to ensure that no other threads can modify the text while you are processing it.

\texttt{editor:with-buffer-locked}  

\texttt{editor:with-buffer-locked}  

\begin{verbatim}
editor:with-buffer-locked (buffer &key for-modification check-file-modification block-interrupts)
 &body body => values
\end{verbatim}

Evaluates \texttt{body} while holding the lock in \texttt{buffer}. At most one thread can lock a buffer at a time and the macro waits until it can claim the lock.

If \texttt{for-modification} is non-nil (the default), the contents of \texttt{buffer} can be modified by \texttt{body}. If \texttt{for-modification} is \texttt{nil}, the contents of \texttt{buffer} cannot be modified until \texttt{body} returns and trying to do so from within \texttt{body} will signal an error. If the buffer is read-only and \texttt{for-modification} is non-nil, then an edi-
tor:editor-error is signalled. The status of the lock can be changed to for-modification (see editor:change-buffer-lock-for-modification). If the buffer is read-only, an editor:editor-error occurs if for-modification is t.

The macro editor:with-buffer-locked can be used recursively, but if the outermost use passed nil as the value of for-modification, then inner uses cannot pass non-nil as the value of for-modification, unless editor:change-buffer-lock-for-modification is used to change the lock status.

If check-file-modification is non-nil (the default) and the buffer is associated with a file and has not already been modified, then the modification time of the file is compared to the time that the file was last read. If the file is newer than the buffer, then the user is asked if they want to re-read the file into the buffer, and if they do then the file is re-read and the operations aborts. Otherwise, there is no check for the file being newer than the buffer.

If block-interrupts is non-nil, the body is evaluated with interrupts blocked. This is useful if the buffer may be modified by an interrupt function, or some interrupt function may end up waiting for another thread that may wait for the buffer lock, which would cause a deadlock. The default is not to block interrupts.

Note that using a non-nil value for block-interrupts is not the same as using the without-interrupts or without-preemption macros. It just stops the current thread from calling interrupt functions, so other threads might run while the body is being evaluated.

The values returned are those of body.

editor:with-point-locked

editor:with-point-locked (point &key for-modification check-file-modification block-interrupts errorp)

&body body => values

Evaluates body while holding the lock in the buffer that is associated with point. In addition, the macro checks that point is valid and this check is
6.3 Programming the editor

atomic with respect to calls to the function `editor:delete-point`. The values of `for-modification`, `check-file-modification` and `block-interrupts` have the same meanings as for `editor:with-buffer-locked`.

The value of `errorp` determines the behavior when `point` is not valid. If `errorp` is non-nil, an error is signaled, otherwise `nil` is returned without evaluating `body`. The point may be invalid because it does not reference any buffer (that is, it has been deleted), or because its buffer was changed by another thread while the current thread was attempting to lock the buffer.

The values returned are those of `body`, or `nil` when `errorp` is `nil` and `point` is not valid.

**editor:change-buffer-lock-for-modification**

Function

### editor:change-buffer-lock-for-modification

`editor:change-buffer-lock-for-modification buffer &key check-file-modification force-modification => result`

Changes the status of the lock in the buffer `buffer` to allow modification of the text. `buffer` must already be locked for non-modification by the current thread (that is, it must be dynamically within a `editor:with-buffer-locked` or `editor:with-point-locked` form with `for-modification` `nil`).

- **buffer**: An editor buffer.
- **check-file-modification**: A boolean.
- **force-modification**: A boolean.
- **result**: `:buffer-not-locked`, `:buffer-out-of-date` or `:buffer-not-writable`.

If `check-file-modification` is non-nil, the same test as described for `editor:with-buffer-locked` is performed, and if the file has been modified then `:buffer-out-of-date` is returned without changing anything (it does not prompt the user to re-read the file).

The default value of `check-file-modification` is `t`. 
force-modification controls what happens if the buffer is read-only. If force-modification is nil, the function returns :buffer-not-writable and does nothing. If it is non-nil, the status is changed. The buffer remains read-only.

result is nil if the status of the locking was changed to for-modification, or if the status of the buffer lock was already for-modification. Otherwise, result is a keyword indicating why the status could not be changed. When result is non-nil, the status of the locking remains unchanged.

The returned value can be one of:

:buffer-not-locked
   The buffer is not locked by the current thread.

:buffer-not-writable
   The buffer is not writable, and force-modification is nil.

:buffer-out-of-date
   The file that is associated with the buffer was modified after it was read into the editor, the buffer is not modified, and check-file-modification is non-nil.

6.3.3.2 Buffer operations

editor:*buffer-list*
   Variable
   Contains a list of all the buffers in the editor.

editor:current-buffer
   Function
   editor:current-buffer
   Returns the current buffer.

editor:buffer-name
   Function
   editor:buffer-name buffer
   Returns the name of buffer.
6.3 Programming the editor

**editor:window-buffer**

*editor:window-buffer* `window`

Returns the buffer currently associated with `window`.

**editor:buffers-start**

*editor:buffers-start* `buffer`

Returns the starting point of `buffer`.

**editor:buffers-end**

*editor:buffers-end* `buffer`

Returns the end point of `buffer`.

**editor:buffer-point**

*editor:buffer-point* `buffer`

Returns the current point in `buffer`.

**editor:use-buffer**

*editor:use-buffer* `buffer` &body `forms`

Makes `buffer` the current buffer during the evaluation of `forms`.

**editor:buffer-from-name**

*editor:buffer-from-name* `name`

Returns the buffer called `name` (which should be a string). If there is no buffer with that name, `nil` is returned.

**editor:make-buffer**

*editor:make-buffer* `name` &key `modes`
Creates a new buffer called name. The argument modes is a list of modes for
the new buffer. The default value for modes is Fundamental. The newly-cre-
ated buffer is returned.

**editor:goto-buffer**

editor:goto-buffer buffer in-same-window

Makes buffer the current buffer. If buffer is currently being shown in a win-
dow then the cursor is moved there. If buffer is not currently in a window
and in-same-window is non-nil then it is shown in the current window, oth-
wise a new window is created for it.

### 6.3.4 Points

Locations within a buffer are recorded as editor:point objects. Each point
remembers a character position within the buffer and all of the editor functions
that manipulate the text of a buffer locate the text using one or more point
objects (sometimes the current point).

A point’s kind controls what happens to the point when text in the buffer is
inserted or deleted.

:temporary points are for cases where you need read-only access to the buffer. They are like GNU Emacs "points". They have a lower overhead than the other kinds of point and do not need to be explicitly deleted, but do not use them in cases where you make a point, insert or delete text and then use the point again, since they do not move when the text is changed. Also, do not use them in cases where more than one thread can modify their buffer without locking the buffer first (see “Buffer locking” on page 171)

:before-insert and :after-insert points are for cases where you need to
make a point, insert or delete text and still use the point afterwards. They are
like GNU Emacs "markers". The difference between these two kinds is what
happens when text is inserted. For a point at position n from the start of the
buffer, inserting \( len \) characters will leave the point at either position \( n \) or \( n+len \) according to the following table.

Table 6.1 Editor point positions after text insertion

<table>
<thead>
<tr>
<th>kind</th>
<th>Insert at &lt; ( n )</th>
<th>Insert at = ( n )</th>
<th>Insert at &gt; ( n )</th>
</tr>
</thead>
<tbody>
<tr>
<td>:before-insert</td>
<td>( n+len )</td>
<td>( n )</td>
<td>( n )</td>
</tr>
<tr>
<td>:after-insert</td>
<td>( n+len )</td>
<td>( n+len )</td>
<td>( n )</td>
</tr>
</tbody>
</table>

When text is deleted, :before-insert and :after-insert points are treated the same: points \( \leq \) the start of the deletion remain unchanged, points \( \geq \) the end of the deletion are moved with the text and points within the deleted region are automatically deleted and cannot be used again.

All points with kind other than :temporary are stored within the data structures of the editor buffer so they can be updated when the text changes. A point can be removed from the buffer by \texttt{editor:delete-point}, and point objects are also destroyed if their buffer is killed.

**editor:point-kind**

\texttt{editor:point-kind point}

Returns the kind of the point, which is :temporary, :before-insert or :after-insert.

**editor:current-point**

\texttt{editor:current-point}

Returns the current point. See also \texttt{editor:buffer-point}.

**editor:current-mark**

\texttt{editor:current-mark &optional pop-pno-error-p}

Returns the current mark. If \texttt{pop-p} is \texttt{t}, the mark ring is rotated so that the previous mark becomes the current mark. If no mark is set and \texttt{no-error-p} is \texttt{t}, \texttt{nil} is returned; otherwise an error is signalled. The default for both of these optional arguments is \texttt{nil}.
Advanced Features

**editor:set-current-mark**

Function

`editor:set-current-mark point`

Sets the current mark to be `point`.

**editor:point<**

Function

`editor:point< point1 point2`

Returns non-nil if `point1` is before `point2` in the buffer.

**editor:point<=**

Function

`editor:point<= point1 point2`

Returns non-nil if `point1` is before or at the same offset as `point2` in the buffer.

**editor:point>**

Function

`editor:point> point1 point2`

Returns non-nil if `point1` is after `point2` in the buffer.

**editor:point>=**

Function

`editor:point>= point1 point2`

Returns non-nil if `point1` is after or at the same offset as `point2` in the buffer.

**editor:copy-point**

Function

`editor:copy-point point &optional kind new-point`

Makes and returns a copy of `point`. The argument `kind` can take the value `:before`, `:after`, or `:temporary`. If `new-point` is supplied, the copied point is bound to that as well as being returned.
### 6.3 Programming the Editor

**editor:delete-point**

*Function*

`editor:delete-point point`

Deletes the point `point`.

This should be done to any non-temporary point which is no longer needed.

**editor:move-point**

*Function*

`editor:move-point point new-position`

Moves `point` to `new-position`, which should itself be a point.

**editor:start-line-p**

*Function*

`editor:start-line-p point`

Returns `t` if `point` is immediately before the first character in a line, and `nil` otherwise.

**editor:end-line-p**

*Function*

`editor:end-line-p point`

Returns `t` if `point` is immediately after the last character in a line, and `nil` otherwise.

**editor:same-line-p**

*Function*

`editor:same-line-p point1 point2`

Returns `t` if `point1` and `point2` are on the same line, and `nil` otherwise.

**editor:save-excursion**

*Macro*

`editor:save-excursion &rest body`

Saves the location of the point and the mark and restores them after completion of `body`. This restoration is accomplished even when there is an abnormal exit from `body`. 
editor:with-point

\texttt{editor:with-point point-bindings \&rest body}

point-bindings is a list of bindings, each of the form \texttt{(var point \{kind\})}. Each variable \texttt{var} is bound to a new point which is a copy of the point \texttt{point} though possibly with a different kind, if \texttt{kind} is supplied. If \texttt{kind} is not supplied, then the new point has \texttt{kind : temporary}.

The forms of \texttt{body} are evaluated within the scope of the point bindings, and then the points in each variable \texttt{var} are deleted, as if by \texttt{editor:delete-point}. Each point \texttt{var} is deleted even if there was an error when evaluating \texttt{body}.

The main reason for using \texttt{with-point} to create non-temporary points is to allow \texttt{body} to modify the buffer while keeping these points up to date for later use within \texttt{body}.

6.3.5 The echo area

editor:message

\texttt{editor:message string \&rest args}

A message is printed in the Echo Area. The argument \texttt{string} must be a string, which may contain formatting characters to be interpreted by \texttt{format}. The argument \texttt{args} consists of arguments to be printed within the string.

editor:clear-echo-area

\texttt{editor:clear-echo-area \&optional string force}

Clears the Echo Area. The argument \texttt{string} is then printed in the Echo Area. If \texttt{force} is non-nil, the Echo Area is cleared immediately, with no delay. Otherwise, there may be a delay for the user to read any existing message.
6.3.6 Editor errors

Many editor commands and functions signal an error on failure (using editor:editor-error as described below). This causes the current operation to be aborted.

In many cases, the user will not want the operation to abort completely if one of the editor commands it uses is not successful. For example, the operation may involve a search, but some aspects of the operation should continue even if the search is not successful. To achieve this, the user can catch the editor:editor-error using a macro such as handler-case.

For example, one part of an application might involve moving forward 5 forms. If the current point cannot be moved forward five forms, generally the Editor would signal an error. However, this error can be caught. The following trivial example shows how a new message could be printed in this situation, replacing the system message.

```
(editor:defcommand "Five Forms" (p)
  "Tries to move the current point forward five forms, printing out an appropriate message on failure."
  "Tries to move the current point forward five forms."
  (handler-case
    (editor:forward-form-command 5)
    (editor:editor-error (condition)
      (editor:message "could not move forward five")))
)
```

**editor:editor-error**

Function

**editor:editor-error string &rest args**

By default this prints a message in the Echo Area, sounds a beep, and exits to the top level of LispWorks, aborting the current operation. The argument string must be a string, which is interpreted as a control string by format. As with editor:message, args can consist of arguments to be processed within the control string.

The behavior is affected by break-on-editor-error.
6.3.7 Files

**editor:find-file-buffer**

`editor:find-file-buffer pathname &optional check-function`

Returns a buffer associated with the file `pathname`, reading the file into a new buffer if necessary. The second value returned is `T` if a new buffer is created, and `nil` otherwise. If the file already exists in a buffer, its consistency is first checked by means of `check-function`. If no value is supplied for `check-function`, `editor:check-disk-version-consistent` is used.

**editor:fast-save-all-buffers**

`editor:fast-save-all-buffers &optional ask`

Saves all modified buffers which are associated with a file. If `ask` is non-nil then confirmation is asked for before saving each buffer. If `ask` is not set, all buffers are saved without further prompting.

Unlike the editor command `Save All Files` this function can be run without any window interaction. It is thus suitable for use in code which does not intend to allow the user to leave any buffers unsaved, and from the console if it is necessary to save buffers without re-entering the full window system.

**editor:check-disk-version-consistent**

`editor:check-disk-version-consistent pathname buffer`

Checks that the date of the file `pathname` is not more recent than the last time `buffer` was saved. If `pathname` is more recent, the user is prompted on how to proceed. Returns `T` if there is no need to read the file from disk and `nil` if it should be read from disk.

**editor:buffer-pathname**

`editor:buffer-pathname buffer`
Returns the pathname of the file associated with buffer. If no file is associated with buffer, nil is returned.

### 6.3.7.1 File encodings in the editor

An editor buffer ideally should have an appropriate external format (or encoding) set before you write it to a file. Otherwise an external format specified in the value of the editor variable output-format-default is used. If the value of output-format-default is not an external-format specifier, then the external format is chosen similarly to the way cl:open does it. By default this chosen external format will be the Windows code page on Microsoft Windows, and Latin-1 on other platforms.

When using the Editor tool, use Set External Format to set interactively the external format for the current buffer, or set Preferences... > Environment > File Encodings > Output (which in turn sets the editor variable output-format-default) to provide a global default value.

In an application which writes editor buffers to file, you can do this to set the external format of a given buffer:

(setf (editor:buffer-external-format buffer) ef-spec)

You can also set a global default external format for editor buffers:

(setf (editor:variable-value 'editor:output-format-default :global) ef-spec)

Then ef-spec will be used when a buffer itself does not have an external format.

### 6.3.7.2 Non base-char errors

If your buffer contains a cl:extended-char char then Latin-1 and other encodings which support only cl:base-char are not appropriate. Attempts to save the buffer using such external formats will signal an error 'char is not of type BASE-CHAR'. Set the external format to one which includes char, or delete char from the buffer before saving.
6.3.7.3 Choosing the encoding to use

You may want a file which is Unicode UCS-2 encoded (external format :unicode), UTF-8 encoding (:utf) or a language-specific encoding such as :shift-jis or :gbk. Or you may want an Latin-1 encoded file, in which case you could pass :latin-1-safe.

6.3.8 Inserting text

**editor:insert-string**

`editor:insert-string point string &optional start end`

Inserts `string` at `point` in the current buffer. The arguments `start` and `end` specify the indices within `string` of the substring to be inserted. The default values for `start` and `end` are 0 and (length `string`) respectively.

**editor:kill-ring-string**

`editor:kill-ring-string &optional index`

Returns either the topmost string on the kill ring, or the string at `index` places below the top when `index` is supplied.

The editor kill ring stores the strings copied by the editor, in order to allow using them later.

**editor:points-to-string**

`editor:points-to-string start end`

Returns the string between the points `start` and `end`. 
6.3 Programming the editor

6.3.9 Indentation

**editor:*indent-with-tabs*** Variable

Controls whether indentation commands such as **Indent** and **Indent Form** insert whitespace using \(#\)Space or \(#\)Tab characters when changing the indentation of a line.

The initial value is **nil**, meaning that only the \(#\)Space character is inserted.

A true value for **editor:*indent-with-tabs*** causes the indentation commands to insert \(#\)Tab characters according to the value of **spaces-for-tab** and then pad with \(#\)Space characters as needed.

6.3.10 Lisp

**editor:*find-likely-function-ignores*** Variable

Contains a list of symbols likely to be found at the beginning of a form (such as **apply**, **funcall**, **defun**, **defmethod**, **defgeneric**).

**editor:*source-found-action*** Variable

This variable determines how definitions found by the commands **Find Source**, **Find Source for Dspec** and **Find Tag** are shown. The value should be a list of length 2.

The first element controls the positioning of the definition: when **t**, show it at the top of the editor window; when a non-negative fixnum, position it that many lines from the top; and when **nil**, position it at the center of the window.

The second element can be **:highlight**, meaning highlight the definition, or **nil**, meaning do not highlight it.

The initial value of **source-found-action*** is (**nil** :**highlight**).
6.3.11  Movement

editor:line-end  

Function

Moves point to be located immediately before the next newline character, or the end of the buffer if there are no following newline characters.

editor:line-start  

Function

Moves point to be located immediately after the previous newline character, or the start of the buffer if there are no previous newline characters.

editor:character-offset  

Function

Moves point forward n characters. If n is negative, point moves back n characters.

editor:word-offset  

Function

Moves point forward n words. If n is negative, point moves back n words.

editor:line-offset  

Function

Moves point n lines forward, to a location to-offset characters into the line. If n is negative, point moves back n lines. If to-offset is nil (its default value), an attempt is made to retain the current offset. An error is signalled if there are not n further lines in the buffer.
6.3 Programming the editor

**editor:form-offset**

Function

`editor:form-offset point n &optional form depth`

Moves point forward `n` Lisp forms. If `n` is negative, point moves back `n` forms. If `form` is `t` (its default value) then atoms are counted as forms, otherwise they are ignored. Before point is moved forward `n` forms, it first jumps out `depth` levels. The default value for `depth` is 0.

6.3.12 Prompting the user

The following functions can be used to prompt for some kind of input, which is generally typed into the Echo Area.

The following keyword arguments are common to a number of prompting functions.

:must-exist Specifies whether the value that is input by the user must be an existing value or not. If :must-exist is non-nil, the user is prompted again if a non-existent value is input.

:default Defines the default value that is selected if an empty string is input.

:default-string Specifies the string that may be edited by the user (with `Insert Parse Default`).

:prompt Defines the prompt that is written in the Echo Area. Most prompting functions have a default prompt that is used if no value is supplied for :prompt.

:help Provides a help message that is printed if the user types "?".

**editor:prompt-for-file**

Function

`editor:prompt-for-file &key direction must-exist create-directories default default-string prompt help`

Prompts for a file name, and returns a pathname.
You can specify `direction :input` (when expecting to read the file) or `direction :output` (when expecting to write the file). This controls the default value of `must-exist`, which is false for `direction :output` and true otherwise.

`:create-directories`

If `create-directories` is true, then the user is prompted to create any missing directories in the path she enters. The default is false for `direction :output` and true otherwise.

See above for an explanation of the other arguments.

**editor:prompt-for-buffer**

Function

`editor:prompt-for-buffer &key prompt must-exist default default-string help`

Prompts for a buffer name, and returns the buffer. See above for an explanation of the keywords.

The default value of `must-exist` is `t`. If `must-exist` is `nil` and the buffer does not exist, it is created.

**editor:prompt-for-integer**

Function

`editor:prompt-for-integer &key prompt must-exist default help`

Prompts for an integer. See above for an explanation of the keywords.

**editor:prompt-for-string**

Function

`editor:prompt-for-string &key prompt default default-string clear-echo-area help`

Prompts for a string. No checking is done on the input. The keyword `clear-echo-area` controls whether or not the echo area is cleared (that is, whether the text being replaced is visible or not). The default for this keyword is `t`. See above for an explanation of the remaining keywords.
6.3 Programming the editor

**editor:prompt-for-variable**

```lisp
editor:prompt-for-variable &key must-exist prompt default default-string help
```

Prompts for an editor variable. See above for an explanation of the keywords. The default value of `must-exist` is `t`.

6.3.13 In-place completion

**editor:complete-with-non-focus**

```lisp
editor:complete-with-non-focus complete-func &key extract-func skip-func insert-func
```

Performs a non-focus completion at the editor current point.

`complete-func` should be a function designator with signature:

```lisp
complete-func string &optional user-arg => result
```

`string` should be a string to complete. `user-arg` is the second return value of `extract-func`, if this is not `nil`. `result` should be a list of items to be displayed in the list panel of the non-focus window.

`extract-func` must be a function designator with signature

```lisp
extract-func point => string, user-arg
```

`point` should be a `Point` object

`extract-func` needs to move `point` to the beginning of the text that will be replaced if the user confirms. It should return two values: `string` is the string to complete, and `user-arg` can be any Lisp object. `string` is passed to the function `complete-func`, and if `user-arg` is non-nil it is also passed.

The default value of `extract-func` is a function which searches backwards until it finds a non-alphanumeric character, or the beginning of the buffer. It then moves its `point` argument forward to the next character. The function returns its first value `string` the string between this and the original location of the point, and it returns `nil` as the second value `user-arg`.

`skip-func`, if supplied, must be a function designator with signature
skip-func point

point should be a Point object

point will be used as the end of the region to replace by the completion. At
the call to skip-func, the point is located at the same place as the point that
was passed to extract-func (after it moved). skip-func needs to move point
forward to the end of the text that should be replaced when the user wants
to do the completion. If skip-func is not supplied, the end point is set to the
current point.

insert-func, if supplied, must be a function designator with signature

insert-func result string user-arg => string-to-use

result is the item selected by the user, string is the original string that was
returned by extract-func, and user-arg is the second value returned by
extract-func (regardless of whether this value is nil). It must return a
string, string-to-use, which is inserted as the the completion.

If insert-func is not supplied, the completion item is inserted. If it is not a
string it is first converted by prinl-to-string.

When editor:complete-with-non-focus is called, it makes a copy of the
current point and passes it to extract-func. It then copies this point and
positions it either using skip-func or the current point. These two points
define the text to be replaced. editor:complete-with-non-focus then
calls complete-func, and use the result to raise a non-focus window next to
the current point. The interaction of this window is as described in Lisp-
Works CAPI User Guide.

6.3.14 Variables

editor:define-editor-variable  Function

editor:define-editor-variable name value &optional documentation

Defines an editor variable.

name Symbol naming the variable.

value The value to assign to the variable.
6.3 Programming the editor

mode A string naming a mode.
documentation A documentation string.

The macro `editor:define-editor-variable` defines a global editor variable. There is only one global value, so repeated uses of `editor:define-editor-variable` overwrite each other.

`editor:define-editor-variable` gives a readable value of defining a variable, and is recognized by the LispWorks source code location system. However variables can also be defined dynamically by calling `(setf editor:variable-value)`. Variable values may be accessed by `editor:variable-value`.

A variable has only one string of documentation associated with it. `editor:variable-value` overwrites the existing documentation string, if there is any. You can see the documentation by the command `Describe Editor Variable`. It can be accessed programmatically by `editor:editor-variable-documentation`.

Note: for backwards compatibility `name` can also be a string, which is converted to a symbol by uppercasing, replacing `#\Space` by `#\-`, and interning in the EDITOR package. This may lead to clashes and so you should use a symbol for `name`, not a string.

**editor:define-editor-mode-variable**

Function

`editor:define-editor-mode-variable name mode value &optional documentation`

Defines an editor variable in the specified mode.

mode A string naming a mode.

name, value As for `editor:define-editor-variable`.

documentation As for `editor:define-editor-variable`, except that `editor:define-editor-mode-variable` installs the documentation only if the editor variable does not already have any documentation.

`editor:define-editor-mode-variable` defines a variable in the specified mode. There is one value per variable per mode.
**editor:define-editor-mode-variable** gives a readable value of defining a variable in a mode, and is recognized by the LispWorks source code location system. However mode variables can also be defined dynamically by calling (**setf editor:variable-value**). Mode variable values may be accessed by **editor:variable-value**.

**editor:editor-variable-documentation**

**editor:editor-variable-documentation** *editor-variable-name*

*editor-variable-name*

A symbol naming an editor variable.

Returns the documentation associated with the editor variable, if any.

**Note:** For backwards compatibility a string *editor-variable-name* is also accepted, as described for **editor:define-editor-variable**.

**editor:variable-value**

**editor:variable-value** *name* &optional *kind* *where*

Returns the value of the editor variable *name*, where *name* is a symbol. An error is signalled if the variable is undefined. The argument *kind* can take the value **:current**, **:buffer**, **:global** or **:mode**. The default value of *kind* is **:current**.

When *kind* is **:current** the argument *where* should be **nil** (the default, meaning the current buffer) or an editor buffer object or the name of a buffer. The variable value for the specified buffer is returned or (if there is no current buffer) then the global variable value is returned.

*kind* can also be **:buffer**, and then *buffer* should be an editor buffer object.

For example, the code given below will, by default, return the value **:ask-user**.

```lisp
(editor:variable-value
  'editor:add-newline-at-eof-on-writing-file)
```
The value of variables may also be altered using this function. For example, the code given below will allow buffers to be saved to file without any prompt for a missing newline.

```
(setf
    (editor:variable-value
        'editor:add-newline-at-eof-on-writing-file)
    nil)
```

### editor:variable-value-if-bound

**Function**

`editor:variable-value-if-bound name &optional kind where`

Returns the value of the variable `name`. If the variable is not bound, `nil` is returned. The arguments are as for `editor:variable-value`.

### editor:buffer-value

**Function**

`editor:buffer-value buffer name &optional errorp`

Accesses the value of the editor variable `name` in the buffer specified by `buffer`.

`name` should be a symbol and `buffer` should be a point object or a buffer object.

If the editor variable is undefined and `errorp` is true, an error is signalled. If the variable is undefined and `errorp` is false, `nil` is returned. The default value of `errorp` is `nil`.

### 6.3.15 Windows

### editor:current-window

**Function**

`editor:current-window`

Returns the current window.

### editor:redisplay

**Function**

`editor:redisplay`
Redisplays any window that appears to need it. In general, the contents of a window may not be redisplayed until there is an event to provoke it.

**Note:** `editor:redisplay` will update a modified editor buffer only when that buffer is the `editor:current-buffer`. Take care to call `editor:redisplay` in an appropriate context.

**editor:window-text-pane**

Returns the `capi:editor-pane` associated with an editor window.

### 6.3.16 Examples

#### 6.3.16.1 Example 1

The following simple example creates a new editor command called **Current Line**.

```
(editor:defcommand "Current Line" (p) "Computes the line number of the current point and
prints it in the Echo Area"
"Prints the line number of the current point"
(let* ((cpoint (editor:current-point))
       (svpoint (editor:copy-point cpoint))
       (count 0))
  (editor:beginning-of-buffer-command nil)
  (loop
    (if (editor:point> cpoint svpoint)
      (return))
    (unless (editor:next-line-command nil)
      (return))
    (incf count))
  (editor:move-point cpoint svpoint)
  (editor:message "Current Line Number: ~S " count))
```

#### 6.3.16.2 Example 2

This example creates a new editor command called **Goto Line** which moves the current point to the specified line number.
(editor:defcommand "Goto Line" (p)
  "Moves the current point to a specified line number. The number can either
  be supplied via the prefix argument, or, if this is nil, it is prompted for."
  "Moves the current point to a specified line number."
  (let ((line-number
    (or p (editor:prompt-for-integer
      :prompt "Line number: "
      :help "Type in the number of the line to go to")))
    (editor:beginning-of-buffer-command nil)
    (editor:next-line-command line-number)))

6.3.16.3 Example 3

The following example illustrates how text might be copied between buffers. First, string is set to all the text in from-buf. This text is then copied to the end of to-buf.

(defun copy-string (from-buf to-buf)
  (let ((string (editor:points-to-string
    (editor:buffers-start from-buf)
    (editor:buffers-end from-buf)))
    (editor:insert-string (editor:buffers-end to-buf) string)))

To test this example, two buffers named t1 and t2 should be created. Then, to copy all the text from t1 to the end of t2:

(copy-string (editor:buffer-from-name "t1")
  (editor:buffer-from-name "t2"))

6.4 Editor source code

The section does not apply to LispWorks Personal Edition.

LispWorks comes with source code for the editor, which you can refer to when adding editor extensions.

6.4.1 Contents

The directory lib/6-0-0-0/src/editor/ contains most of the source files of the LispWorks editor. Some low-level source code is not distributed.
6.4.2 Source location

To enable location of editor definitions by Find Source and related commands, configure LispWorks as described under "Finding source code" in the LispWorks User Guide and Reference Manual.

6.4.3 Guidelines for use of the editor source code

Some care is needed when working with the supplied editor source code, to ensure that you do not compromise the IDE or introduce a dependancy on a particular release of LispWorks.

In particular please note:

- The editor source code may not match the compiled code in the LispWorks image exactly, for example if editor patches have been loaded.
- Modifications to the EDITOR package definition are not allowed.
- Redefining existing definitions is not recommended. It is better to define a new command to do what you want. If you find a bug or have a useful extension to an existing definition then please let us know.
- Do not rely on the expansion of exported macros.
- If you use any internal (that is, not exported) EDITOR symbols, please tell us, so we can consider how to support your requirements. In addition, some internal macros have been removed from the LispWorks image and these should not be used.
Abbrev

An abbrev (abbreviation) is a user defined text string which, when typed into a buffer, may be expanded into another string using Abbrev Mode. Typing can therefore be saved by defining short strings to be expanded into frequently used longer words or phrases.

Abbrevs should not be confused with the abbreviated symbol completion implemented by the command Abbreviated Complete Symbol.

Abbrev Mode

Abbrev mode is a minor mode which allows abbrevs to be automatically expanded when typed into a buffer.

Auto-Fill Mode

Auto-fill mode is a minor mode which allows lines to be broken between words at the right margin automatically as the text is being typed. This means that return does not have to be pressed at the end of each line to simulate filling.

Auto-Saving

Auto-saving is the automatic, periodic backing-up of the file associated with the current buffer.
Backup
When a file is explicitly saved in the editor, a backup is automatically made by writing the old contents of the file to a backup before saving the new version of the file. The name of the backup file is that of the original file followed by a ~ character.

Binding
A binding is made up of one or more key sequences. A command may have a default binding associated with it, which executes that command. Bindings provide a quick and easy way to execute commands.

Buffer
A buffer is a temporary storage area used by the editor to hold the contents of a file while the process of editing is taking place.

Case Conversion
Case conversion means changing the case of text from lower to upper case and vice versa.

Completion
Completion is the process of expanding a partial or abbreviated name into the full name. Completion can used for expanding symbols, editor command names, filenames and editor buffer names.

Control Key
The Control key (Ctrl) is used as part of many key sequences. Ctrl must be held down while pressing the required character key.

Ctrl Key
See Control Key.

Current
The adjective current is often used to describe a point, buffer, mark, paragraph, and similar regions of text, as being the text area or item on which relevant commands have an effect. For example, the current buffer is the buffer on which most editor commands operate.
Cursor
The cursor is the rectangle (in Emacs emulation) or vertical bar (in other emulations) seen in a buffer which indicates the position of the current point within that buffer.

Customization
Customization means making changes to the way the editor works. The editor can be customized both in the short and long term to suit the user’s requirements. Short term customization involves altering the way the editor works for the duration of an editing session by using standard editor commands, while long term customization involves programming the editor.

Default
A default is the value given to an argument if none is specified by the user.

Deleting
Deleting means removing text from the buffer without saving it. The alternative is killing.

Echo Area
The Echo Area is a buffer used to display and input editor information. Commands are typed into this buffer and editor produced messages are displayed here.

Emulation
The LispWorks Editor can behave like GNU Emacs, or like a typical editor on the KDE/Gnome platform. Keys, cursors, behavior with selected text and other functionality differs. We use the term KDE/GnomeEmulation to denote this alternate behavior.

Escape Key
The Escape key (Esc) has its own functionality but is mostly used in Emacs emulation in place of the Meta key when no such key exists on a keyboard. Esc must be typed before pressing the required character key.
**Extended Command**

Most editor commands can be invoked explicitly by using their full command names, preceded by the `Meta+X` key sequence. A command issued in such a way is known as an extended command.

**Fill Prefix**

The fill prefix is a string which is ignored when filling takes place. For example, if the fill prefix is `;;;`, then these characters at the start of a line are skipped over when the text is re-formatted.

**Filling**

Filling involves re-formatting text so that each line extends as far to the right as possible without any words being broken or any text extending past a predefined right-hand column.

**Global Abbrev**

A global abbrev is an abbrev which can be expanded in all major modes.

**History Ring**

The history ring records Echo Area commands so that they can easily be repeated.

**Incremental Search**

An incremental search is a search which is started as soon as the first character of the search string is typed.

**Indentation**

Indentation is the blank space at the beginning of a line. Lisp, like many other programming languages, has conventions for the indentation of code to make it more readable. The editor is designed to facilitate such indentation.

**Insertion**

Insertion is the process of inputting text into a buffer.
Keyboard Macro
A keyboard macro allows a sequence of editor commands to be turned into a single operation. Keyboard macros are only available for the duration of an editing session.

Key Sequence
A key sequence is a sequence of characters used to issue, or partly issue, an editor command. A single key sequence usually involves holding down one of two specially defined modifier keys (that is Ctrl and Meta), while at the same time pressing another key.

Killing
Killing means removing text from a buffer and saving it in the kill ring, so that the text may be recovered at a later date. The alternative is deleting.

Kill Ring
The kill ring stores text which has been killed, so that it may be recovered at a later date. Text can be re-inserted into a buffer by yanking. There is only one kill ring for all buffers so that text can be copied from one buffer to another.

Major Mode
Major modes govern how certain commands behave. They adapt a few editor commands so that their use is more appropriate to the text being edited. For example, the concept of indentation is radically different in Lisp mode and Fundamental mode. Each buffer is associated with one major mode.

Mark
A mark stores the location of a point so that it may be used for reference at a later date. More than one mark may be associated with a single buffer and saved in a mark ring.
**Mark Ring**

The mark ring stores details of marks, so that previously defined marks can be accessed. The mark ring works like a stack, in that marks are pushed onto the ring and can only be popped off on a "last in first out" basis. Each buffer has its own mark ring.

**Meta Key**

On most PC keyboards this key is synonymous with the Alt key. However, there are many different types of keyboard, and the Meta key may not be marked with "Alt" or "Meta". It may be marked with a special character, such as a diamond, or it may be one of the function keys — try F11.

In Emacs emulation, Meta must be held down while pressing the required character key. As some keyboards do not have a Meta key, the Escape (Esc) key can be used in place of Meta.

On Cocoa, you can configure "Meta" by choosing Preferences... > Environment > Emulation.

**Minor Mode**

The minor modes determine whether or not certain actions take place. For example, when abbrev mode is on, abbrevs are automatically expanded when typed into a buffer. Buffers may possess any number of minor modes.

**Mode**

Each buffer has two modes associated with it: a major mode and a minor mode. A buffer must have one major mode, but can have zero or more minor modes associated with it. Major modes govern how certain commands behave, while minor modes determine whether or not certain actions take place.

**Mode Abbrev**

A mode abbrev is an abbrev which is expanded only in predefined major modes.
Mode Line
At the bottom of each buffer is a mode line that provides information concerning that buffer. The information displayed includes name of the buffer, major mode, minor mode and whether the buffer has been modified or not.

Newline
Newline is a whitespace character which terminates a line of text.

Overwrite Mode
Overwrite mode is a minor mode which causes each character typed to replace an existing character in the text.

Page
A page is the region of text between two page delimiters. The ASCII key sequence Ctrl+L constitutes a page delimiter (as it starts a new page on most line printers).

Pane
A pane is the largest portion of an editor window, used to display the contents of a buffer.

Paragraph
A paragraph is defined as the text within two paragraph delimiters. A blank line constitutes a paragraph delimiter. The following characters at the beginning of a line are also paragraph delimiters: Space Tab @ - ' )

Prefix Argument
A prefix argument is an argument supplied to a command which sometimes alters the effect of that command, but in most cases indicates how many times that command is to be executed. This argument is known as a prefix argument as it is supplied before the command to which it is to be applied. Prefix arguments sometimes have no effect on a command.
Point

A point is a location in a buffer where editor commands take effect. The current point is generally the location between the character indicated by the cursor and the previous character (that is, it actually lies between two characters). Many types of commands (moving, inserting, deleting) operate with respect to the current point, and indeed move that point.

Recursive Editing

Recursive editing occurs when you are allowed to edit text while an editor command is executing.

Region

A region is the area of text between the mark and the current point. Many editor commands affect only a specified region.

Register

Registers are named slots in which locations and regions can be saved for later use.

Regular Expression Searching

A regular expression (regexp) allows the specification of a search string to include wild characters, repeated characters, ranges of characters, and alternatives. Strings which follow a specific pattern can be located, which makes regular expression searches very powerful.

Replacing

Replacing means substituting one string for another.

Saving

Saving means copying the contents of a buffer to a file.

Scrolling

Scrolling means slightly shifting the text displayed in a pane either upwards or downwards, so that a different portion of the buffer is displayed.
Searching
Searching means moving the current point to the next occurrence of a specified string.

Sentence
A sentence begins wherever a paragraph or previous sentence ends. The end of a sentence is defined as consisting of a sentence terminating character followed by two spaces or a newline. The following characters are sentence terminating characters: . ? !

Tag File
A tag file is one which contains information on the location of Lisp function definitions in one or more files. For each file in a defined system, the tag file contains a relevant file name entry, followed by names and positions of each defining form in that file. This information is produced by the editor and is required for some definition searches.

Transposition
Transposition involves taking two units of text and swapping them round so that each occupies the others former position.

Undoing
Commands that modify text in a buffer can be undone, so that the text reverts to its state before the command was invoked.

Undo Ring
An undo ring is used to hold details of modifying commands so that they can be undone at a later date. The undo ring works like a stack, in that commands are pushed onto the ring and can only be popped off on a "last in first out" basis.

Variable (Editor)
Editor variables are parameters which affect the way that certain commands operate.
Whitespace

Whitespace is any consecutive run of the whitespace characters \texttt{Space}, \texttt{Tab} or \texttt{Newline}.

Window

A window is an object used by the window manager to display data. When the editor is called up, an editor window is created and displayed.

Window Ring

A window ring is used to hold details of all windows currently open.

Word

A word is a continuous string of alphanumeric characters (that is, the letters A–Z and numbers 0–9). In most modes, any character which is not alphanumeric is treated as a word delimiter.

Yanking

Yanking means inserting a previously killed item of text from the kill ring at a required location. This is often known as \textit{pasting}. 
Symbols
# files 25
? Help on Parse 104
~ files 25, 34

A
Abbrev Expand Only 98
Abbrev Mode 96
abbrev mode 93, 96
Abbreviated Complete Symbol 132
abbreviation
add global 97
add global expansion 97
add mode 96
add mode expansion 97
append to file 100
delete all 99
delete global 99
delete mode 98
edit 100
editor definition 96
expand 98
list 99
read from file 101
save to file 100
undo last expansion 98
abbreviation commands 96
abbrev-pathname-defaults 100
Abort Recursive Edit 108
aborted editor commands 11, 16
aborting processes 11, 16
Add Global Word Abbrev 97
Add Mode Word Abbrev 96
add-newline-at-eof-on-writing-file 30
Append Next Kill 53
Append to File 29
Append to Word Abbrev File 100
Application Builder tool 116
Apropos 145
Apropos Command 20
argument
listing for function 133
prefix 24
attribute
description 21
listing with apropos 20
Auto Fill Linefeed 68
Auto Fill Mode 67
Auto Fill Return 68
Auto Fill Space 67
Auto Save Toggle 32
auto-fill mode 67, 93
auto-fill-space-indent 68
auto-save file 32
auto-save-checkpoint-frequency 33
auto-save-clean-up-checkpoints 33
auto-save-filename-pattern 33
auto-save-key-count-threshold 33

B
Back to Indentation 64
Backup File 30
backup file 30, 34
backup-filename-pattern 34
backup-filename-suffix 34
backups-wanted 34
Backward Character 38
Backward Form 136
Backward Kill Form 137
Backward Kill Line 52
Backward Kill Sentence 52
Backward List 139
Backward Paragraph 40
Backward Search 80
Backward Sentence 40
Backward Up List 140
Backward Word 38
base-char type 183
Beginning of Buffer 42
Beginning of Defun 119
Beginning of Line 38
Beginning Of Parse 105
Beginning of Parse or Line 106
Bind Key 108
Bind String to Key 109
binding
editor definition 10
binding keys 108
bind-key 164
bind-string-to-key 165
Bottom of Window 42
Break Definition 128
Break Definition on Exit 128
Break Function 128
Break Function on Exit 128
breaking processes 17
break-on-editor-error 115
buffer
changed definitions in 132
circulate 69
compile 153
compile changed definitions 154
compile if necessary 153
create 70
editor definition 6
evaluate 150
evaluate changed definitions 150
file options 36
insert 71
kill 37, 69
list 70
mark whole 46
modified check 71
move to beginning 42
move to end 43
new 70
not modified 72
print 71
read only 71
rename 71
revert 35
save 28
search all 81
select 68
select in other window 69
select previous 69
set package 147
buffer 170
Buffer Changed Definitions 132
buffer commands 68
buffer functions 170, 193
Buffer Not Modified 72
buffer-from-name 175
*buffer-list* 174
buffer-name 174
buffer-pathname 182
buffer-point 175
buffers and windows 113
buffers-end 175
buffers-start 175
buffer-value 193
bug
reporting 115
Build Application 115
button
mouse bindings in editor 114
C
calling editor functions 167
Capitalize Region 59
Capitalize Word 58
case conversion commands 57
case-replace 88
CD 111
Center Line 66
change-buffer-lock-for-modification 173
character
backward 38
delete expanding tabs 49
delete next 49
delete previous 49
forward 38
insert with overwrite 61
overwrite previous 61
transposition 59
character-offset 186
Check Buffer Modified 71
check-disk-version-consistent 182
Index

Circulate Buffers 69
class
describe 134
Class Browser tool 134
clear-echo-area 180
colors
Lisp syntax 118
command
abort 16
completion 10, 18, 103
description 20, 21
execution 9, 18, 167
history 22
key sequence for 23
key sequences 23
listing with apropos 20
repetition 11, 24
shell 110
commands
abbreviation 96
aborting commands 11, 16
aborting processes 11, 16
buffer 68
case conversion 57
compilation 147, 152
cut and paste 13
deleting text 13, 48
echo area 103
editing Lisp programs 117
editor variable 107
evaluation 147, 148, 151
file handling 12, 25
filling 64
help 14, 18
indentation 61
inserting text 12, 53
key binding 108
keyboard macro 101
killing text 13, 48
Lisp comment 140
Lisp documentation 145
Lisp form 136
Lisp function and definition 119
Lisp list 139
movement 12, 37
overwriting 60
pages 75
parentheses 143, 145
recursive editing 107
register 90
replacing 77
running shell from editor 110
searching 77
transposition 59
undoing 13, 56
window 72
comment
create 141
kill 142
move to 141
comment commands 140
comment-begin 143
comment-column 143
comment-end 143
comments
inserting 141
comment-start 143
compilation commands 147, 152
compilation messages
finding the source code 155
compile
buffer 153
buffer changed definitions 154
buffer if necessary 153
changed definitions 154
file 153
form 152
region 153
system 155
system changed definitions 155
Compile Buffer 153
Compile Buffer Changed Definitions 154
Compile Buffer File 153
Compile Changed Definitions 154
Compile Defun 152
Compile File 153
Compile Region 153
Compile System 155
Compile System Changed
Definitions 155
compile-buffer-file-confirm 154
Complete Field 104
Complete Input 103
Complete Symbol 131
complete-with-non-focus 189
completion
dynamic word 56
in-place 189
of abbreviated symbols 132
of commands 10, 18, 103
of filenames 56
of symbols 131
configuration files 160, 163
Confirm Parse 104
Index

Continue Tags Search 125
Copy to Cut Buffer 113
Copy to Register 91
copy-point 178
Count Lines Page 76
Count Lines Region 47
Count Matches 86
Count Occurrences 86
Count Words Region 46
Create Buffer 70
Create Tags Buffer 123
Ctrl key 9
Ctrl+I Abort Recursive Edit 108
Ctrl+I Function Arglist Display 133
Ctrl+A Beginning of Line 38
Ctrl+A Beginning Of Parse or Line 106
Ctrl+B Backward Character 38
Ctrl+B Echo Area Backward Character 105
Ctrl+C Ctrl+C Insert Selected Text 107
Ctrl+d Delete Next Character 49
Ctrl+e End of Line 39
Ctrl+f Forward Character 38
Ctrl+g, abort current command 16
Ctrl+h A Apropos 14
Ctrl+h A Apropos Command 20
Ctrl+h B Describe Bindings 23
Ctrl+H C What Command 20
Ctrl+h Ctrl+d Document Command 21
Ctrl+H Ctrl+k Document Key 22
Ctrl+h Ctrl+v Document Variable 22
Ctrl+h Ctrl+g Describe Command 14, 20
Ctrl+H G Generic Describe 21
Ctrl+H Help 19
Ctrl+H K Describe Key 14, 21
Ctrl+h L What Lossage 22
Ctrl+H V Describe Editor Variable 22
Ctrl+H W Where Is 23
Ctrl+k Kill Line 52
Ctrl+L Refresh Screen 74
Ctrl+N Next Line 39
Ctrl+o Open Line 55
Ctrl+P Insert Parse Default 106
Ctrl+P Previous Line 39
Ctrl+q Quoted Insert 55
Ctrl+r Return Default 107
Ctrl+R Reverse Incremental Search 78
Ctrl+s Esc Forward Search 79
Ctrl+s Incremental Search 77
Ctrl+Shift-_ Undo 13, 57
Ctrl+Shift+A Function Argument List 134
Ctrl+Shift+b Compile Buffer 153
Ctrl+Shift+c Compile Defun 152
Ctrl+Shift+d Function Documentation 146
Ctrl+Shift+e Evaluate Region 149
Ctrl+Shift+m Macroexpand Form 138
Ctrl+Shift+r Compile Region 153
Ctrl+Space Set Mark 44
Ctrl+t Transpose Characters 59
Ctrl+u Kill Parse 106
Ctrl+u Set Prefix Argument 24
Ctrl+v Scroll Window Down 40
Ctrl+w Kill Region 53
Ctrl+x - Inverse Add Global Word Abbrev 97
Ctrl+x & Search Files Matching Patterns 82
Ctrl+x ( Define Keyboard Macro 102
Ctrl+x ) End Keyboard Macro 102
Ctrl+x * Search Files 82
Ctrl+x + Add Global Word Abbrev 97
Ctrl+X . Set Fill Prefix 66
Ctrl+X / Point to Register 90
Ctrl+X ; Set Comment Column 140
Ctrl+X { Previous Page 75
Ctrl+X } Next Page 75
Ctrl+x - Check Buffer Modified 71
Ctrl+x 0 Delete Window 73
Ctrl+x 1 Delete Next Window 73
Ctrl+x 2 New Window 72
Ctrl+x b Select Buffer 68
Ctrl+x c Go Back 48
Ctrl+X Ctrl+a Add Mode Word Abbrev 96
Ctrl+X Ctrl+b List Buffers 70
Ctrl+X Ctrl+c Save All Files and Exit 30
Ctrl+X Ctrl+e Evaluate Last Form 149
Ctrl+X Ctrl+f Wfind File 26
Ctrl+X Ctrl+h Inverse Add Mode Word Abbrev 97
Ctrl+X Ctrl+i Indent Rigidly 63
Ctrl+X Ctrl+l Lowercase Region 58
Ctrl+X Ctrl+o Delete Blanks Lines 50
Ctrl+X Ctrl+p Mark Page 76
Ctrl+X Ctrl+q Toggle Buffer Read-Only 71
Ctrl+X Ctrl+s Save File 28
Ctrl+X Ctrl+t Transpose Lines 60
Index

Ctrl+X Ctrl+U Uppercase Region 58
Ctrl+X Ctrl+V Find Alternate File 27
Ctrl+X Ctrl+W Write File 29
Ctrl+X Ctrl+X Exchange Point and Mark 45
Ctrl+X Delete Backward Kill Sentence 52
Ctrl+X E Last Keyboard Macro 102
Ctrl+X F Set Fill Column 65
Ctrl+X G Insert Register 91
Ctrl+X H Mark Whole Buffer 46
Ctrl+X I Insert File 36
Ctrl+X I Insert File 36
Ctrl+X J Jump to Register 90
Ctrl+X K Kill Buffer 69
Ctrl+X L Count Lines Page 76
Ctrl+X M Select Go Back 48
Ctrl+X O Next Ordinary Window 72
Ctrl+X P Next Window 72
Ctrl+X Q Go Forward 48
Ctrl+X Q Keyboard Macro Query 102
Ctrl+X S Save All Files 28
Ctrl+X Tab Indent Rigidly 63
Ctrl+X X Copy to Register 91
Ctrl+Y Un-Kill 14, 53
Ctrl-C Ctrl-C Interrupt Shell Subjob 112
Ctrl-C Ctrl-D Shell Send Eof 113
Ctrl-C Ctrl-Z Stop Shell Subjob 112
current point
editor definition 7
current-buffer 174
current-mark 177
current-package 147
current-point 177
current-window 193
customising
editor 163
editor commands 163 indentation of Lisp forms 163, 166
key bindings 160, 163, 164
cut and paste commands 13

D
debugger
using in editor 150
default
binding 10
external format 27, 30
prefix argument 24
default-auto-save-on 33
default-buffer-element-type 70
default-modes 93
default-search-kind 83
defcommand macro 168
Defindent 119
Define Keyboard Macro 102
Define Word Abbrevs 101
define-editor-mode-variable 191
define-editor-variable 190
definition
break 128
disassemble 155
editing 119
find 120
find buffer changes 132
searching for 120
trace 127
trace inside 127
untrace 127
defmode function 94
Delete All Word Abbrevs 99
Delete Blank Lines 50
DELETE Delete Previous Character 49
DELETE Echo Area Delete Previous Character 106
Delete File 36
Delete File and Kill Buffer 37
Delete Global Word Abbrev 99
Delete Horizontal Space 50
Delete Indentation 63
Delete Key Binding 109
Delete Matching Lines 80
Delete Mode Word Abbrev 98
Delete Next Character 49
Delete Next Window 73
Delete Non-Matching Lines 81
Delete Previous Character 49
Delete Previous Character Expanding Tabs 49
Delete Region 50
Delete Selection Mode 56
Delete Window 73
delete-point 179
deleting text 49
deleting text commands 13, 48
deletion
editor definition 48
of selection 56
of surrounding form 137
delimiter
sentence 9
Describe Bindings 23
Describe Class 134
Describe Command 20
Index

Describe Editor Variable 22
Describe Generic Function 135
Describe Key 21
Describe Method Call 135
Describe Symbol 145
Describe System 135
Diff 89
Diff Ignoring Whitespace 90
directory
   change 111
   query replace 88
   search 81
Directory Query Replace 88
Directory Search 81
Disassemble Definition 155
Do Nothing 110
Document Command 21
Document Key 22
Document Variable 22
documentation commands 145
double-quotes
   inserting 139
Down Comment Line 142
Down List 140
dspec
   documentation 146
Dynamic Completion 56

E
echo area
   complete text 103
   completing commands in 103
   deleting and inserting text in 106
editor definition 103
   help on parse 104
   movement in 105
   next command 105
   previous command 104
   prompting the user 187
   repeating commands in 104
   terminate entry 104
Echo Area Backward Character 105
Echo Area Backward Word 105
echo area commands 103
Echo Area Delete Previous Character 106
echo area functions 180, 194
Echo Area Kill Previous Word 106
Edit Callee 130
Edit Callers 130
Edit Editor Command 122
Edit Recognized Source 155
Edit Word Abbrevs 100

editor
   customising 163
delete-region-command 50
   programming 166
editor commands
   Abbrev Expand Only 98
   Abbrev Mode 96
   Abbreviated Complete Symbol
      Meta+I 132
   Abort Recursive Edit Ctrl+X 108
   Add Global Word Abbrev Ctrl+X + 97
   Add Mode Word Abbrev Ctrl+X
      Ctrl+A 96
   Append Next Kill Meta+Ctrl+W 53
   Append to File 29
   Append to Word Abbrev File 100
   Apropos 145
   Apropos Command Ctrl+H A 20
   Auto Fill Linefeed LINEFEED 68
   Auto Fill Mode 67
   Auto Fill Return RETURN 68
   Auto Fill Space SPACE 67
   Auto Save Toggle 32
   Back to Indentation Meta+M 64
   Backup File 30
   Backward Character Ctrl+B 38
   Backward Form Meta+Ctrl+B 136
   Backward Kill Form
      Meta+Ctrl+Delete 137
   Backward Kill Line 52
   Backward Kill Sentence Ctrl+X
      Delete 52
   Backward List Meta+Ctrl+F 139
   Backward Paragraph Meta+I 40
   Backward Search 80
   Backward Sentence Meta+A 40
   Backward Up List Meta+Ctrl+U 140
   Backward Word Meta+B 38
   Beginning of Buffer Meta<- 42
   Beginning of Defun Meta+Ctrl+A 119
   Beginning of Line Ctrl+A 38
   Beginning Of Parse Meta<- 105
   Beginning of Parse or Line Ctrl+A 106
   Bind Key 108
   Bind String to Key 109
   Bottom of Window 42
   Break Definition 128
   Break Definition on Exit 128
   Break Function 128
   Break Function on Exit 128
   Buffer Changed Definitions 132
   Buffer Not Modified Meta+Shift+- 72
Build Application 115
Capitalize Region 59
Capitalize Word Meta+C 58
CD 111
Center Line 66
Check Buffer Modified Ctrl+X - 71
Circulate Buffers Meta+Ctrl+Shift+L 69
Compile Buffer Changed Definitions 154
Compile Buffer File 153
Compile Changed Definitions 154
Compile Defun Ctrl+Shift+C 152
Compile File 153
Compile Region Ctrl+Shift+R 153
Compile System 155
Compile System Changed Definitions 155
Complete Field SPACE 104
Complete Input TAB 103
Complete Symbol Meta+Ctrl+I 131
Confirm Parse RETURN 104
Continue Tags Search Meta+, 125
Copy to Cut Buffer 113
Copy to Register Ctrl+X X 91
Count Lines Page Ctrl+X L 76
Count Lines Region 47
Count Matches 86
Count Occurrences 86
Count Words Region 46
Create Buffer 70
Create Tags Buffer 123
Defindent 119
Define Keyboard Macro Ctrl+X 102
Define Word Abbrevs 101
Delete All Word Abbrevs 99
Delete Blank Lines Ctrl+X Ctrl+O 50
Delete File 36
Delete File and Kill Buffer 37
Delete Global Word Abbrev 99
Delete Horizontal Space Meta\\ 50
Delete Indentation Meta+Shift+" 63
Delete Key Binding 109
Delete Matching Lines 80
Delete Mode Word Abbrev 98
Delete Next Character Ctrl+D 49
Delete Next Window Ctrl+X 73
Delete Non-Matching Lines 81
Delete Previous Character DELETE 49
Delete Previous Character Expanding Tabs 49
Delete Region 50
Delete Selection Mode 56
Delete Window Ctrl+X 0 73
Describe Bindings Ctrl+H B 23
Describe Class 134
Describe Command Ctrl+H D 20
Describe Editor Variable Ctrl+H V 22
Describe Generic Function 135
Describe Key Ctrl+H K 21
Describe Method Call 135
Describe Symbol 145
Describe System 135
Diff 89
Diff Ignoring Whitespace 90
Directory Query Replace 88
Directory Search 81
Disassemble Definition 155
Do Nothing 110
Document Command Ctrl+H Ctrl+D 21
Document Key Ctrl+H Ctrl+K 22
Document Variable Ctrl+H Ctrl+V 22
Down Comment Line Meta+\ 142
Down List Meta+Ctrl+D 140
Dynamic Completion Meta+/ 56
Echo Area Backward Character Ctrl+B 105
Echo Area Backward Word Meta+B 105
Echo Area Delete Previous Character DELETE 106
Echo Area Kill Previous Word Meta+Delete 106
Edit Callees 130
Edit Callers 130
Edit Editor Command 122
Edit Recognized Source 155
Edit Word Abbrevs 100
End Keyboard Macro Ctrl+X ) 102
End of Buffer Meta>> 43
End of Defun Meta+Ctrl+X 102
End of Line Ctrl+E 39
Evaluate Buffer 150
Evaluate Buffer Changed Definitions 150
Evaluate Changed Definitions 151
Evaluate Defun In Listener 151
Evaluate Defun Meta+Ctrl+X 148
Evaluate Expression Escape+Escape 149
Evaluate Last Form In Listener 151
Evaluate Last Form Ctrl+X Ctrl+E 149
Evaluate Region In Listener 152
Index

Evaluate Region Ctrl+Shift+E 149
Evaluate System Changed Definitions 151
Exchange Point and Mark Ctrl+X Ctrl+X 45
Exit Recursive Edit Meta+Ctrl+Z 108
Expand File Name Meta+Tab 56
Extended Command Meta+X 10, 18
Extract List Meta+X 138
Fill Paragraph Meta+Q 65
Fill Region Meta+G 65
Find Alternate File Ctrl+X Ctrl+V 27
Find Command Definition 122
Find File 26
Find Mismatch 145
Find Source for Dspec 121
Find Source Meta+. 120
Find Unbalanced Parentheses 145
Find Unwritable Character 31
Forward Character Ctrl+F 38
Forward Form Meta+Ctrl+F 136
Forward Kill Form Meta+Ctrl+X 137
Forward Kill Sentence Meta+K 52
Forward List Meta+Ctrl+N 139
Forward Paragraph Meta+] 40
Forward Search Ctrl+S Esc 79
Forward Up List 140
Forward Word Meta+F 38
Function Arglist Display Ctrl+\ 133
Function Arglist Meta+= 133
Function Argument List Ctrl+Shift+A 134
Function Documentation Ctrl+Shift+D 146
Fundamental Mode 92
Generic Describe Ctrl+H G 21
Get Register 91
Go Back Ctrl+X C 48
Go Forward Ctrl+X F 48
Goto Line 39
Goto Page 75
Goto Point 43
Help Ctrl+H 19
Help on Parse ? 104
Illegal 109
Incremental Search Ctrl+S 77
Indent for Comment Meta+; 141
Indent Form Meta+Ctrl+Q 136
Indent New Comment Line Meta+J or Meta+Newline 142
Indent New Line 64
Indent or Complete Symbol 131
Indent Region Meta+Ctrl+\ 62
Indent Rigidity 63
Indent Rigidity Ctrl+X Tab, Ctrl+X Ctrl+X 63
Indent Selection or Complete Symbol TAB 131
Indent TAB 62
Insert () 143
Insert Buffer 71
Insert Cut Buffer 113
Insert Double Quotes For Selection Meta+ 139
Insert File Ctrl+X I 36
Insert Multi Line Comment For Selection Meta+# 141
Insert Page Directory 76
Insert Parentheses For Selection Meta+* 144
Interrupt Shell Subjob Ctrl+C 112
Inverse Add Global Word Abbrev Ctrl+X 97
Inverse Add Mode Word Abbrev Ctrl+X Ctrl+H 97
Jump to Register Ctrl+X J 90
Jump to Saved Position 90
Just One Space Meta+Space 50
Keyboard Macro Query Ctrl+X G 102
Kill Backward Up List 137
Kill Buffer Ctrl+X K 69
Kill Comment Meta+Ctrl+; 142
Kill Line Ctrl+K 52
Kill Next Word Meta+) 51
Kill Parse Ctrl+U 106
Kill Previous Word Meta+Delete 52
Kill Region Ctrl+W 53
Kill Register 90
Last Keyboard Macro Ctrl+X E 102
Line to Top of Window 42
Lisp Insert ) 144
Lisp Mode 93
List Buffers Ctrl+X Ctrl+B 70
List Callees 129
List Callers 129
List Definitions 122
List Definitions For Dspec 123
Index

Shell 110
Shell Command Meta+! 110
Shell Send Eof Ctrl-C Ctrl-D 113
Show Documentation for Dspec 146
Show Documentation
Meta+Ctrl+Shift+A 146
Show Paths From 130
Show Paths To 129
Show Variable 107
Skip Whitespace 43
Stop Shell Subjob Ctrl-C Ctrl-Z 112
System Query Replace 88
System Search 82
Tags Query Replace 125
Tags Search 124
Text Mode 92
Toggle Auto Save 32
Toggle Breakpoint 156
Toggle Buffer Read-Only Ctrl+X Ctrl+Q 71
Toggle Count Newlines 74
Toggle Error Catch 150
Toggle Buffer Read-Only 138
Top of Window 42
Trace Definition 127
Trace Definition Inside Definition 127
Trace Function 126
Trace Function Inside Definition 127
Transpose Characters Meta+Ctrl+Shift+T 59
Transpose Forms Meta+Ctrl+T 138
Transpose Lines Ctrl+X Ctrl+T 60
Transpose Regions 60
Transpose Words Meta+T 59
Undefine 157
Undefine Buffer 157
Undefine Command 157
Undefine Region 158
Undo Ctrl+Shift+Z 57
Unexpand Last Word 98
Un-Kill Ctrl+Y 53
Untrace Definition 127
Untrace Function 127
Up Comment Line Meta+P 142
Uppercase Region Ctrl+X Ctrl+U 58
Uppercase Word Meta+U 58
View Page Directory 76
View Source Search 122
Visit File 27
Visit Other Tags File 126
Visit Tags File 125
Walk Form Meta+Shift+H 138
Wfind File Ctrl+X Ctrl+F 26
What Command Ctrl+H C 20
What Cursor Position 43
What Line 39
What Lossage Ctrl+W L 22
Where Is Ctrl+W W 23
Where is Point 43
Word Abbrev Apropos 99
Word Abbrev Prefix Point Meta+’ 98
Write File Ctrl+X Ctrl+W 29
Write Region 29
Write Word Abbrev File 100
desktop errors
debugging 115
desktop functions
bind-key 164
bind-string-to-key 165
buffer-from-name 175
buffer-name 174
buffer-pathname 182
buffer-point 175
buffers-end 175
buffers-start 175
buffer-value 193
change-buffer-lock-for-modification 173
copy-point 178
current-buffer 174
current-mark 177
current-point 177
current-window 193
define-editor-mode-variable 191
define-editor-variable 190
define-editor-mode-variable 191
define-editor-variable 190
define-error 181
define-editor-mode-variable 192
define-editor-variable 192
define-editor-mode-variable 192
define-editor-variable 192
define-editor-mode-variable 192
define-editor-character 193
define-editor-mode-variable 192
define-editor-character 193
define-editor-mode-variable 192
define-editor-character 193
define-editor-mode-variable 192
define-editor-character 193
define-editor-mode-variable 192
define-editor-character 193
end-line-p 179
fast-save-all-buffers 182
find-file-buffer 182
find-file-buffer 182
form-offset 187
goto-buffer 176
insert-string 184
kill-ring-string 184
line-end 186
line-offset 186
line-start 186
make-buffer 175
message 180
move-point 179
point< 178
Index

point<= 178
point> 178
point>= 178
point-kind 177
points-to-string 184
process-character 167
prompt-for-buffer 188
prompt-for-file 187
prompt-for-integer 188
prompt-for-string 188
prompt-for-variable 189
redisplay 193
same-line-p 179
set-current-markt 178
set-interrupt-keys 165
setup-indent 166
start-line-p 179
variable-value 192
variable-value-if-bound 193
window-buffer 175
window-text-pane 194
word-offset 186
editor macros
save-excursion 179
use-buffer 175
with-buffer-locked 171
with-point 180
with-point-locked 171, 172
editor package 166
editor source code 195
Editor tool 130
editor types
buffer 170
point 176
editor variable 107
editor variables
abbrevpathname-defaults 100
add-newline-at-eof-on-writing-file 30
auto-fill-space-indent 68
auto-save-checkpoint-frequency 33
auto-save-cleanup-checkpoints 33
auto-save-filename-pattern 33
auto-save-key-count-threshold 33
backup-filename-pattern 34
backup-filename-suffix 34
backups-wanted 34
break-on-editor-error 115
case-replace 88
comment-begin 143
comment-columnn 143
comment-end 143
comment-start 143
compile-buffer-file-confirm 154
current-package 147
default-auto-save-on 33
default-buffer-element-type 70
default-modes 93
default-search-kind 83
evaluate-defvar-action 148
fill-column 65
fill-prefix 66
highlight-matching-parens 144
input-format-default 27
output-format-default 30
prefix-argument-default 24
prompt-regexp-string 112
region-query-size 47
revert-buffer-confirm 36
save-all-files-confirm 29
scroll-overlap 41
shell-cd-regexp 111
shell-pop-regexp 112
shell-push-regexp 111
spaces-for-tab 62
undo-ring-size 57
editor-error 181
teditor-variable-documentation 192
encoding
default 27, 30
setting 31
unwritable character 31
unwritable characters 32
End Keyboard Macro 102
End of Buffer 43
End of Defun 119
End of Line 39
end-line-p 179
error
catching evaluation 150
editor 181
error functions 181
Escape key 9
EscapeEvaluateExpression 149
evaluate
buffer 150
buffer changed definition 150
changed definitions 151
defvar 149
expression 149
file 150
form 148, 151
last form 149, 151
region 149, 152
Index

system changed definitions 151
Evaluate Buffer 150
Evaluate Buffer Changed Definitions 150
Evaluate Changed Definitions 151
Evaluate Defun 148
Evaluate Defun In Listener 151
Evaluate Expression 149
Evaluate Last Form 149
Evaluate Last Form In Listener 151
Evaluate Region 149
Evaluate Region In Listener 152
Evaluate System Changed Definitions 151
evaluate-defvar-action 148
evaluation commands 147, 148, 151
evaluation commands for buffer 147, 148, 151
examples
  programming the editor 194
Exchange Point and Mark 45
execute mode 93
executing editor commands 9, 18
Exit Recursive Edit 108
Expand File Name 56
expression
  evaluate 149
Extended Command 10, 18
extended-char type 183
external format
  default 27, 30
  setting 31
  unwritable character 31
  unwritable characters 32
external formats 183
Extract List 138

F
fast-save-all-buffers 182
file
  auto-saving 32
  backup 30, 34
  compile 153
  delete 36, 37
  editor definition 6
  evaluate 150
  expand name 56
  find alternate 27
  finding 26
  insert into buffer 36
  options for buffer 36
  print 35
  rename 37
  save 28, 30
  set external format 31
  unwritable character 31
  unwritable characters 32
  write 29
file encodings 183
file functions 193
file handling commands 12, 25
filename completion 56
files
  search 82
Fill Paragraph 65
Fill Region 65
fill-column 65
filling commands 64
fill-prefix 66
Find Alternate File 27
Find Command Definition 122
Find File 26
Find Mismatch 145
Find Source 120
Find Source for Dspec 121
Find Tag 123
Find Unbalanced Parentheses 145
Find Unwritable Character 31
find-file-buffer 182
finding editor source code 122
*find-likely-function-ignores* 185
form
  compile 152
  evaluate 148, 151
  evaluate last 149, 151
  indent 136
  kill backwards 137
  kill forwards 137
  macro-expand 138
  mark 136
  move to beginning 136
  move to end 136
  transposition 138
form commands 136
form-offset 187
Forward Character 38
Forward Form 136
Forward Kill Form 137
Forward Kill Sentence 52
Forward List 139
Forward Paragraph 40
Forward Search 79
Forward Sentence 40
Forward Up List 140
Forward Word 38
function
  argument list 133
break 128
describe generic 135
documentation 146
edit callees 130
edit callers 130
editing 119
find definition 120
indentation 119
list callees 129, 130
list callers 129
mark 119
move to beginning 119
move to end 119
trace 126
trace inside 127
untrace 127
Function Arglist 133
Function Arglist Display 133
Function Argument List 134
Function Call Browser tool 129, 130
Function Documentation 146
functions
buffer 170, 193
calling 167
defmode 94
echo area 180, 194
editor error 181
editor, see editor functions
file 193
inserting text 184
Lisp editor 185
movement 186, 193
point 176
prompt 187
search-files 83
variable 190
window 193
Fundamental Mode 92
fundamental mode 92
G
Generic Describe 21
generic function
describe 135
Generic Function Browser tool 135
Get Register 91
global abbreviation
definition 96
Go Back 48
Go Forward 48
Goto Line 39
Goto Page 75
Goto Point 43
goto-buffer 176
H
Help 19
help commands 14, 18
Help on Parse 104
highlight-matching-parens 144
history of commands 22
history ring 104
I
Illegal 109
Incremental Search 77
Indent 62
indent
form 136
Indent for Comment 141
Indent Form 136
Indent New Comment Line 142
Indent New Line 64
Indent or Complete Symbol 131
Indent Region 62
Indent Rigidly 63
Indent Selection 63
Indent Selection or Complete Symbol 131
indentation
customising 163, 166
define for Lisp forms 119
define for Lisp functions 119
delete 63
move back to 64
indentation commands 61
indenting 185
*indent-with-tabs* 185
In-place completion 189
input-format-default 27
Insert 0 143
Insert Buffer 71
Insert Cut Buffer 113
Insert Double Quotes For Selection 139
Insert File 36
Insert Multi Line Comment For Selection 141
Insert Page Directory 76
Insert Parentheses For Selection 144
Insert Parse Default 106
Insert Register 91
Insert Selected Text 107
Insert Word Abbrevs 101
inserting text commands 12, 53
Index

inserting text functions 184
\textit{insert-string} 184
Interrupt Shell Subjob 112
Inverse Add Global Word Abbrev 97
Inverse Add Mode Word Abbrev 97

J
Jump to Register 90
Jump to Saved Position 90
Just One Space 50

K
key
command description 20
Control 9
description 21
Escape 9
Meta 9
key binding 108
customising 160, 163, 164
key sequence
editor definition 9
for command 23
key sequences
for commands 23
keyboard macro
begin definition of 102
terminology 101
for definition of 102
execute 102
name 102
keyboard macro commands 101
Keyboard Macro Query 102
Kill Backward Up List 137
Kill Buffer 69
Kill Comment 142
Kill Line 52
Kill Next Word 51
Kill Parse 106
Kill Previous Word 52
Kill Region 53
Kill Register 90
kill ring 48, 51, 53
rotate 54
killing
editor definition 48
killing text 51
killing text commands 13, 48
\textit{kill-ring-string} 184
L
Last Keyboard Macro 102
\textit{line}
beginning 38
centre 66
count for page 76
count for region 47
delete blank 50
delete matching 80
delete non-matching 81
end 39
goto 39
indent new 64
indentation 131
kill 52
kill backward 52
length 65
list matching 80
move to top of window 42
next 39
open new 54
previous 39
transposition 60
what line 39
line count 76
\textit{Line to Top of Window} 42
\textit{line-end} 186
\textit{LINEFEED} Auto Fill Linefeed 68
\textit{line-offset} 186
\textit{line-start} 186
Lisp
editor commands 117
Lisp comment commands 140
Lisp documentation commands 145
Lisp editor functions 185
Lisp form commands 136
Lisp \textit{Insert}) 144
Lisp list commands 139
Lisp Mode 93
Lisp mode 92
LispWorks IDE tools
Application Builder 116
Class Browser 134
Editor 130
Function Call Browser 129, 130
Generic Function Browser 135
Listener 50, 51, 93, 112, 151
Output Browser 51
Process Browser 17
Search Files 82, 83
Shell 110, 112, 113
Symbol Browser 145
list
  extract 137
kill backward up 137
move down one level 140
move to end 139, 140
move to start 139, 140
List Buffers 70
List Callees 129
List Callers 129
list commands 139
List Definitions 122
List Definitions For Dspec 123
List Matching Lines 80
List Registers 91
List Unwritable Characters 32
List Word Abbrevs 99
Listener tool 50, 51, 93, 112, 151
Load File 150
locations 47
Lowercase Region 58
Lowercase Word 57
M
  macro
    keyboard 101
  Macroexpand Form 138
macro-expansion 138
macros
  defcommand 168
major mode
  editor definition 8, 91
Make Word Abbrev 97
make-buffer 175
man Unix command 23
manual
  on-line editor 21, 22
Manual Entry 23
mark
  editor definition 7
exchange with point 45
form 136
move current point to 45
paragraph 46
pop 45
sentence 46
set 44
word 45
  See also locations
Mark Defun 119
Mark Form 136
Mark Page 76
Mark Paragraph 46
mark ring 44
Mark Sentence 46
Mark Whole Buffer 46
Mark Word 45
message 180
Meta key 9
Meta-! Shell Command 110
Meta* Insert Double Quotes For
Selection 139
Meta+# Insert Multi Line Comment For
Selection 141
Meta+( Insert Parentheses For
Selection 144
Meta+) Move Over ) 144
Meta+, Continue Tags Search 125
Meta++. Find Source 120
Meta+/ Dynamic Completion 56
Meta+; Indent for Comment 141
Meta+< Beginning of Buffer 42
Meta+< Beginning Of Parse 105
Meta++ Function Arglist 133
Meta>> End of Buffer 43
Meta? Find Tag 123
Meta@ Mark Word 45
Meta[ Backward Paragraph 40
Meta\ Delete Horizontal Space 50
Meta{} Forward Paragraph 40
Meta` Word Abbrev Prefix Point 98
Meta`A Backward Sentence 40
Meta`B Backward Word 38
Meta`B Echo Area Backward Sentence 105
Meta`C Capitalize Word 58
Meta`Ctrl+.Rotate Active Finders 126
Meta`Ctrl+; Kill Comment 142
Meta`Ctrl@ Mark Form 136
Meta`Ctrl\ Indent Region 62
Meta`Ctrl+\ Beginning of Defun 119
Meta`Ctrl+B Backward Form 136
Meta`Ctrl+C, break gesture 17
Meta`Ctrl+D Down List 140
Meta`Ctrl+Delete Backward Kill
Form 137
Meta`Ctrl+E End of Defun 119
Meta`Ctrl+F Forward Form 136
Meta`Ctrl+H Mark Defun 119
Meta`Ctrl+I Complete Symbol 131
Meta`Ctrl+K Forward Kill Form 137
Meta`Ctrl+L Select Previous
Buffer 69
Meta`Ctrl+N Forward List 139
Meta`Ctrl+P Backward List 139
Meta`Ctrl+Q Indent Form 136
Index

Meta+Ctrl+R  Regexp Reverse Search  86
Meta+Ctrl+S  Regexp Forward Search  86
Meta+Ctrl+Shift+A  Show Documentation  146
Meta+Ctrl+Shift+L  Circulate Buffers  69
Meta+Ctrl+T  Transpose Forms  138
Meta+Ctrl+U  Backward Up List  140
Meta+Ctrl+W  Append Next Kill  53
Meta+Ctrl+X  Evaluate Defun  148
Meta+Ctrl+Z  Exit Recursive Edit  108
Meta+D  Kill Next Word  51
Meta+Delete  Echo Area Kill Previous Word  106
Meta+Delete  Kill Previous Word  52
Meta+E  Forward Sentence  40
Meta+F  Forward Word  38
Meta+G  Fill Region  65
Meta+H  Mark Paragraph  46
Meta+I  Abbreviated Complete Symbol  132
Meta+J  Indent New Comment Line  142
Meta+K  Forward Kill Sentence  52
Meta+L  Lowercase Word  57
Meta+M  Back to Indentation  64
Meta+N  Down Comment Line  142
Meta+N  Next Parse  105
Meta+Newline  Indent New Comment Line  142
Meta+P  Previous Parse  104
Meta+P  Up Comment Line  142
Meta+Q  Fill Paragraph  65
Meta+Shift+%  Query Replace  87
Meta+Shift+`  Delete Indentation  63
Meta+Shift+-  Buffer Not Modified  72
Meta+Shift+M  Walk Form  138
Meta+Shift+R  Move to Window Line  42
Meta+Space  Just One Space  50
Meta+T  Transpose Words  59
Meta+Tab  Expand File Name  56
Meta+U  Uppercase Word  58
Meta+V  Scroll Window Up  41
Meta+W  Save Region  53
Meta+X  Extended Command  10, 18
Meta+Y  Rotate Kill Ring  54

method call
describe  135
minor mode
editor definition  8, 93
mode
editor definition  8, 91

indentation in  61
mode abbreviation
editor definition  96
mode line
editor definition  6
modes
abbrev  93, 96
auto-fill  67, 93
execute  93
fundamental  92
Lisp  92
overwrite  61, 93
shell  92
text  92
mouse
editor bindings  114
Move Over  144
Move to Window Line  42
movement commands  12, 37
locations  47
movement functions  186, 193
move-point  179
MS windows keys
using  159

N
Name Keyboard Macro  102
Negative Argument  25
New Buffer  70
New in LispWorks 6.0
buffer-value  193
Build Application  115
change-buffer-lock-for-modification  173
define-editor-mode-variable  191
define-editor-variable  190
Describe Method Call  135
diff ignoring whitespace  90
documentation  128
editor-variable-documentation  192
Find Unwritable Character  31
Function Arglist Display  133
Indent or Complete Symbol  131
Indent Selection or Complete Symbol  131
Insert Double Quotes For Selection  139
Insert Multi Line Comment For Selection  141
Insert Parentheses For Selection  144
List Unwritable Characters  32
Make Directory  37
Mark Word 45
   Next Breakpoint 156
Previous Breakpoint 156
Toggle Count Newlines 74
New Line 54
New Window 72
newline
   adding to end of file 30
Next Breakpoint 156
Next Line 39
Next Ordinary Window 72
Next Page 75
Next Parse 105
Next Window 72

O
Open Line 55
Output Browser tool 51
output-format-default 30
Overwrite Delete Previous Character 61
Overwrite Mode 61
overwrite mode 61, 93
overwriting commands 60

P
package
   editor 166
      set 147
page
   display first lines 76
      editor definition 75
      goto 75
   insert first lines into buffer 76
   mark 76
   next 75
   previous 75
   page commands 75
   pane
      editor definition 5
Paragraph
   backward 40
      editor definition 9
      fill 65
      forward 40
      mark 46
parentheses
   inserting a pair of 143, 144
   parentheses commands 143
   pending delete 56
Point
   editor definition 7
   exchange with mark 45
   goto 43
   move to window line 42
   position of 43
   save to register 90
   where is 43
point 176
point behavior 176
point functions 176
point ring, see mark ring
Point to Register 90
point< 178
point<= 178
point> 178
point>= 178
point-kind 177
points and text modification 176
points-to-string 184
Pop and Goto Mark 45
Pop Mark 45
prefix
   fill 66
   prefix argument 11, 24
   prefix-argument-default 24
Previous Breakpoint 156
Previous Line 39
Previous Page 75
Previous Parse 104
Previous Window 73
print
   buffer 71
      file 35
      region 47
   Print Buffer 71
   Print File 35
   Print Region 47
process
   breaking 17
   Process Browser tool 17
Process File Options 36
   process-character 167
   programming the editor 166
      calling functions 167
      examples 194
   prompt functions 187
   prompt-for-buffer 188
   prompt-for-file 187
   prompt-for-integer 188
   prompt-for-string 188
   prompt-for-variable 189
   prompt-regexp-string 112
   Put Register 91
Index

Q
Query Replace 87
query replace 87
directory 88
regexp 89
system 88
tags 125
Query Replace Regexp 89
Quote Tab 64
Quoted Insert 55

R
Read Word Abbrev File 101
recursive editing 107
redisplay 193
Re-evaluate Defvar 149
Reevaluate Defvar 149
Refresh Screen 74
regexp
query replace 89
replace 89
Regexp Forward Search 86
Regexp Reverse Search 86
region
append 29
capitalize 59
compile 153
delete 50
determining 45
erator definition 8
evaluate 149, 152
fill 65
get from register 91
indent 62
indent rigidly 63
kill 53
line count 47
lowercase 58
print 47
save 53
transposition 60
uppercase 58
word count 46
write 29
region-query-size 47
register
editor definition 90
get region 91
kill 90
list 91
move to saved position 90
record position 91
save current point to 90
register commands 90
Register to Point 90
regular expression 84
count occurrences of 86
regular expression search 84
Rename Buffer 71
Rename File 37
repeating a command 11, 24
replace
case sensitivity 88
query 87
regexp 89
string 87
Replace Regexp 89
Replace String 87
replacing 87
replacing commands 77
Report Bug 115
Report Manual Bug 115
RETURN Auto Fill Return 68
RETURN Confirm Parse 104
Return Default 107
RETURN New Line 54
Reverse Incremental Search 78
Reverse Search 80
Revert Buffer 35
revert-buffer-confirm 36
ring
history 104
kill 48, 51, 53
mark 44
undo 56
window 72
Room 115
Rotate Active Finders 126
Rotate Kill Ring 54
Run Command 110

S
same-line-p 179
Save All Files 28
Save All Files and Exit 30
Save File 28
Save Position 90
Save Region 53
save-all-files-confirm 29
save-excursion 179
screen
refresh 74
scroll button
  size 74
Scroll Next Window Down 73
Scroll Next Window Up 73
Scroll Window Down 40
Scroll Window Up 41
scroller
  size 74
scroll-overlap 41
search
  all buffers 81
  backward 80
  case sensitivity 83
  directory 81
  files 82
  forward 79
  incremental backward 78
  incremental forward 77
  regexp backward 86
  regexp forward 86
  regular expression 84
  system 82, 83
Search All Buffers 81
Search Files 82
Search Files Matching Patterns 82
Search Files tool 82, 83
Search System 83
search-files function 83
searching 77
searching commands 77
Select Buffer 68
Select Buffer Other Window 69
Select Go Back 48
Select Previous Buffer 69
selection
  indent 63
  indenting 131
Self Insert 55
Self Overwrite 61
sentence
  backward 40
  delimiter 9
  editor definition 8
  forward 40
  kill backward 52
  kill forward 52
  mark 46
  terminator 9
Set Buffer Output 147
Set Buffer Package 147
Set Comment Column 140
Set External Format 31
Set Fill Column 65
Set Fill Prefix 66
Set Mark 44
Set Prefix Argument 24
Set Variable 107
set-current-mark 178
set-interactive-break-gestures
  function 17
set-interrupt-keys 165
setup-indent 166
Shell 110
Shell Command 110
shell command
  from editor 110
shell mode 92
Shell Send Eof 113
Shell tool 110, 112, 113
shell-cd-regexp 111
shell-pop-regexp 112
shell-push-regexp 111
Show Documentation 146
Show Documentation for Dspec 146
Show Paths From 130
Show Paths To 129
Show Variable 107
simple-char type 70
Skip Whitespace 43
source finding
  active finders list 126
dspec 121
editor command 122
editor definitions 196
name 120
tags 123
tags files 124, 125
source recording 120
*source-found-action* 185
space
  delete horizontal 50
  just one 50
SPACE Auto Fill Space 67
SPACE Complete Field 104
spaces-for-tab 62
start-line-p 179
Stop Shell Subjob 112
string
  count occurrences of 86
  insert 184
  replace 87
  search 77
symbol
  apropos 145
Index

browser 145
  completion 131, 132
  describe 145
Symbol Browser tool 145
Syntax coloring 118
system
  compile 155
  compile changed definitions 155
  describe 135
  evaluate changed definitions 151
  query replace 88
  search 82, 83
System Query Replace 88
System Search 82
T
TAB
  for command completion 103
  for indentation 62, 131
  for symbol completion 131
Tab
  for command completion 10, 18
  insert 64
  width 62
TAB Complete Input 103
TAB Indent 62
TAB Indent Selection or Complete Symbol 131
tag
  continue search 125
  create buffer 123
  editor definition 120
  find 123
  query replace 125
  search 124
  visit file 125
Tags Query Replace 125
Tags Search 124
temporary files 34
terminator
  sentence 9
text handling concepts 8
Text Mode 92
text mode 92
Toggle Auto Save 32
Toggle Breakpoint 156
Toggle Buffer Read-Only 71
Toggle Count Newlines 74
Toggle Error Catch 150
Top of Window 42
Trace Definition 127
Trace Definition Inside Definition 127
Trace Function 126
Trace Function Inside Definition 127
tracing functions 126
Transpose Characters 59
Transpose Forms 138
Transpose Lines 60
Transpose Regions 60
Transpose Words 59
transposition commands 59
U
Undefine 157
  undefine
    buffer 157
    command 157
    definition 157
    region 158
Undefine Buffer 157
Undefine Command 157
Undefine Region 158
Undo 57
undo ring 56
  size 57
undoing editor commands 13, 56
undo-ring-size 57
Unexpand Last Word 98
Unix command
  man 23
Un-Kill 53
Untrace Definition 127
Untrace Function 127
Up Comment Line 142
Uppercase Region 58
Uppercase Word 58
use-buffer 175
V
variable
  change value of 107
  description 21, 22
  editor 107
    listing with apropos 20
    show value of 107
  variable functions 190
  variables
    *buffer-list* 174
    *find-likely-function-ignores* 185
    *indent-with-tabs* 185
    *source-found-action* 185
    indenting 185
variable-value 192
variable-value-if-bound 193
Visit Page Directory 76
Visit File 27
Visit Other Tags File 126
Visit Tags File 125

W
Walk Form 138
Wfind File 26
What Command 20
What Cursor Position 43
What Line 39
What Lossage 22
Where Is 23
Where Is Point 43
whitespace
  skip 43
window
  delete 73
  delete next 73
  editor definition 5
  mode line 74
  move line to top of 42
  move to bottom 42
  move to top 42
  new 72
  next 72
  previous 73
  scroll down 40
  scroll next down 73
  scroll next up 73
  scroll overlap 41
  scroll up 41
  scroller 74
  window commands 72
  window functions 193
  window ring 72
window-buffer 175
windows
  and the Editor 113
  copy 113
  paste 113
window-text-pane 194
with-buffer-locked 171
with-point 180
with-point-locked 171, 172
word
  backward 38
  capitalize 58
  count for region 46
dynamic completion 56
editor definition 8
forward 38
kill next 51
kill previous 52
lowercase 57
mark 45
transposition 59
uppercase 58
Word Abbrev Apropos 99
Word Abbrev Prefix Point 98
word-offset 186
Write File 29
Write Region 29
Write Word Abbrev File 100

Y
yank 53
Index