

jade

COLLABORATORS

	<i>TITLE :</i> jade		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY		September 19, 2022	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

1	jade	1
1.1	jade.guide	1
1.2	jade.guide/Copying	2
1.3	jade.guide/Introduction	2
1.4	jade.guide/News	3
1.5	jade.guide/Systems Supported	5
1.6	jade.guide/Editor Concepts	6
1.7	jade.guide/Key Names	7
1.8	jade.guide/Modifiers	8
1.9	jade.guide/Keys	9
1.10	jade.guide/Example Keys	10
1.11	jade.guide/Starting Jade	11
1.12	jade.guide/Invocation	11
1.13	jade.guide/Startup Options	12
1.14	jade.guide/Startup Procedure	14
1.15	jade.guide/Using Jade	15
1.16	jade.guide/Invoking Commands	17
1.17	jade.guide/Command Arguments	17
1.18	jade.guide/The Help System	18
1.19	jade.guide/Loading and Saving Files	19
1.20	jade.guide/Commands To Load Files	20
1.21	jade.guide/Commands To Save Files	20
1.22	jade.guide/File Variables	21
1.23	jade.guide/Backup Files	22
1.24	jade.guide/Auto-Saving Files	23
1.25	jade.guide/Accessing Compressed Files	24
1.26	jade.guide/Other File Commands	24
1.27	jade.guide/Editing Buffers	24
1.28	jade.guide/Moving Around Buffers	26
1.29	jade.guide/Undo	27

1.30	jade.guide/Editing Units	28
1.31	jade.guide/Editing Characters	29
1.32	jade.guide/Editing Words	30
1.33	jade.guide/Editing Expressions	31
1.34	jade.guide/Editing Lines	31
1.35	jade.guide/Cutting And Pasting	32
1.36	jade.guide/Using Blocks	33
1.37	jade.guide/Marking Blocks	34
1.38	jade.guide/Commands on Blocks	34
1.39	jade.guide/Rectangular Blocks	35
1.40	jade.guide/Killing	36
1.41	jade.guide/Searching and Replacing	36
1.42	jade.guide/Regular Expressions	37
1.43	jade.guide/Incremental Search	39
1.44	jade.guide/Global Replace	40
1.45	jade.guide/Query Replace	40
1.46	jade.guide/Editing Modes	41
1.47	jade.guide/Invoking a Mode	43
1.48	jade.guide/Generic mode	43
1.49	jade.guide/C mode	43
1.50	jade.guide/Lisp mode	44
1.51	jade.guide/Asm mode	46
1.52	jade.guide/Text mode	46
1.53	jade.guide/Indented-Text mode	47
1.54	jade.guide/Texinfo mode	47
1.55	jade.guide/Minor Modes	49
1.56	jade.guide/Overwrite mode	49
1.57	jade.guide/Fill mode	50
1.58	jade.guide/Auto-Save mode	50
1.59	jade.guide/Latin-1 mode	50
1.60	jade.guide/Using Buffers	51
1.61	jade.guide/Displaying Buffers	52
1.62	jade.guide/Deleting Buffers	52
1.63	jade.guide/Other Buffer Commands	53
1.64	jade.guide/The Buffer Menu	53
1.65	jade.guide/Using Windows	55
1.66	jade.guide/Creating Windows	56
1.67	jade.guide/Killing Windows	57
1.68	jade.guide/Other Window Commands	57

1.69	jade.guide/Using the Prompt	58
1.70	jade.guide/The Simple Prompt	58
1.71	jade.guide/The Buffer Prompt	59
1.72	jade.guide/Using Marks	59
1.73	jade.guide/Interrupting Jade	60
1.74	jade.guide/Recursive Editing	61
1.75	jade.guide/Character Images	62
1.76	jade.guide/Client Editing	63
1.77	jade.guide/Compiling Programs	64
1.78	jade.guide/Running a Compilation	64
1.79	jade.guide/Finding Errors	65
1.80	jade.guide/Debugging Programs	66
1.81	jade.guide/Using Grep	67
1.82	jade.guide/Keeping ChangeLogs	68
1.83	jade.guide/Info Mode	68
1.84	jade.guide/Shell	70
1.85	jade.guide/Simple Customisation	72
1.86	jade.guide/Programming Jade	73
1.87	jade.guide/Intro	75
1.88	jade.guide/nil and t	76
1.89	jade.guide/The Lisp Reader	77
1.90	jade.guide/Notation	77
1.91	jade.guide/Descriptions	77
1.92	jade.guide/Data Types	79
1.93	jade.guide/Types Summary	79
1.94	jade.guide/Read Syntax	82
1.95	jade.guide/Printed Representation	83
1.96	jade.guide/Equality Predicates	83
1.97	jade.guide/Comparison Predicates	85
1.98	jade.guide/Type Predicates	85
1.99	jade.guide/Garbage Collection	86
1.100	jade.guide/Numbers	87
1.101	jade.guide/Arithmetic Functions	87
1.102	jade.guide/Bitwise Functions	88
1.103	jade.guide/Numeric Predicates	89
1.104	jade.guide/Characters	90
1.105	jade.guide/Sequences	91
1.106	jade.guide/Cons Cells	92
1.107	jade.guide/Lists	93

1.108jade.guide/List Structure	94
1.109jade.guide/Building Lists	95
1.110jade.guide/Accessing List Elements	98
1.111jade.guide/Modifying Lists	99
1.112jade.guide/Association Lists	100
1.113jade.guide/Infinite Lists	101
1.114jade.guide/Vectors	102
1.115jade.guide/Strings	103
1.116jade.guide/Array Functions	105
1.117jade.guide/Sequence Functions	106
1.118jade.guide/Symbols	107
1.119jade.guide/Symbol Syntax	108
1.120jade.guide/Symbol Attributes	108
1.121jade.guide/Oarrays	109
1.122jade.guide/Creating Symbols	110
1.123jade.guide/Interning	111
1.124jade.guide/Property Lists	112
1.125jade.guide/Evaluation	113
1.126jade.guide/Symbol Forms	114
1.127jade.guide/List Forms	114
1.128jade.guide/Function Call Forms	115
1.129jade.guide/Macro Call Forms	116
1.130jade.guide/Special Forms	116
1.131jade.guide/Autoload Forms	116
1.132jade.guide/Self-Evaluating Forms	117
1.133jade.guide/Quoting	117
1.134jade.guide/Control Structures	118
1.135jade.guide/Sequencing Structures	118
1.136jade.guide/Conditional Structures	119
1.137jade.guide/Looping Structures	122
1.138jade.guide/Non-Local Exits	122
1.139jade.guide/Catch and Throw	123
1.140jade.guide/Function Exits	124
1.141jade.guide/Cleanup Forms	125
1.142jade.guide/Errors	125
1.143jade.guide/Variables	126
1.144jade.guide/Local Variables	127
1.145jade.guide/Setting Variables	129
1.146jade.guide/Scope and Extent	130

1.147jade.guide/Buffer-Local Variables	130
1.148jade.guide/Void Variables	133
1.149jade.guide/Constant Variables	133
1.150jade.guide/Defining Variables	134
1.151jade.guide/Functions	135
1.152jade.guide/Lambda Expressions	136
1.153jade.guide/Named Functions	137
1.154jade.guide/Anonymous Functions	138
1.155jade.guide/Predicate Functions	139
1.156jade.guide/Defining Functions	140
1.157jade.guide/Calling Functions	141
1.158jade.guide/Mapping Functions	141
1.159jade.guide/Macros	142
1.160jade.guide/Defining Macros	143
1.161jade.guide/Macro Expansion	143
1.162jade.guide/Compiling Macros	144
1.163jade.guide/Streams	145
1.164jade.guide/Input Streams	146
1.165jade.guide/Output Streams	147
1.166jade.guide/Input Functions	148
1.167jade.guide/Output Functions	149
1.168jade.guide/Loading	151
1.169jade.guide/Load Function	152
1.170jade.guide/Autoloading	153
1.171jade.guide/Features	155
1.172jade.guide/Compiled Lisp	156
1.173jade.guide/Compilation Functions	156
1.174jade.guide/Compilation Tips	157
1.175jade.guide/Disassembly	159
1.176jade.guide/Hooks	160
1.177jade.guide/Functions As Hooks	160
1.178jade.guide/Normal Hooks	160
1.179jade.guide/Standard Hooks	161
1.180jade.guide/Buffers	163
1.181jade.guide/Buffer Attributes	164
1.182jade.guide/Creating Buffers	166
1.183jade.guide/Modifications to Buffers	167
1.184jade.guide/Read-Only Buffers	167
1.185jade.guide/Destroying Buffers	167

1.186jade.guide/Special Buffers	168
1.187jade.guide/The Buffer List	169
1.188jade.guide/The Current Buffer	170
1.189jade.guide/Windows	171
1.190jade.guide/Opening Windows	172
1.191jade.guide/Closing Windows	173
1.192jade.guide/Iconifying Windows	174
1.193jade.guide/Displaying Messages	174
1.194jade.guide/The Current Window	175
1.195jade.guide/Window Font	176
1.196jade.guide/Window Information	177
1.197jade.guide/Rendering	178
1.198jade.guide/Block Marking	179
1.199jade.guide/Positions	180
1.200jade.guide/Position Components	181
1.201jade.guide/The Cursor Position	182
1.202jade.guide/Movement Functions	182
1.203jade.guide/Buffer Extremes	183
1.204jade.guide/Character Movement	184
1.205jade.guide/Word Movement	185
1.206jade.guide/Tab Movement	186
1.207jade.guide/Line Movement	186
1.208jade.guide/Expression Movement	187
1.209jade.guide/Positions and Offsets	187
1.210jade.guide/Marks	188
1.211jade.guide/Mark Components	189
1.212jade.guide/Mark Relocation	189
1.213jade.guide/Mark Residency	190
1.214jade.guide/Creating Marks	191
1.215jade.guide/Altering Marks	192
1.216jade.guide/Moving to Marks	192
1.217jade.guide/Glyph Tables	192
1.218jade.guide/Glyph Table Basics	193
1.219jade.guide/Glyph Positions	194
1.220jade.guide/Creating Glyph Tables	195
1.221jade.guide/Buffer Glyph Tables	195
1.222jade.guide/Input Events	196
1.223jade.guide/Keymaps	197
1.224jade.guide/Types of Keymap	197

1.225jade.guide/Creating Keymaps	198
1.226jade.guide/Binding Keys	198
1.227jade.guide/Key Lookup	199
1.228jade.guide/Prefix Keys	200
1.229jade.guide/Standard Keymaps	201
1.230jade.guide/Event Loop	201
1.231jade.guide/Event Loop Actions	202
1.232jade.guide/Commands	203
1.233jade.guide/Interactive Declarations	204
1.234jade.guide/Prefix Arguments	206
1.235jade.guide/Calling Commands	207
1.236jade.guide/Example Commands	208
1.237jade.guide/Event Loop Info	209
1.238jade.guide/Recursive Edits	209
1.239jade.guide/Reading Events	210
1.240jade.guide/Idle Actions	211
1.241jade.guide/Editing Files	212
1.242jade.guide/Reading Files Into Buffers	213
1.243jade.guide/Writing Buffers	215
1.244jade.guide/Buffer Date Stamps	216
1.245jade.guide/Buffer Modification Counts	217
1.246jade.guide/Making Backups	217
1.247jade.guide/Controlling Auto-Saves	218
1.248jade.guide/Text	219
1.249jade.guide/Buffer Contents	219
1.250jade.guide/Insertion Functions	220
1.251jade.guide/Deletion Functions	221
1.252jade.guide/Kill Functions	222
1.253jade.guide/Transpose Functions	223
1.254jade.guide/Indentation Functions	224
1.255jade.guide/Translation Functions	225
1.256jade.guide/Search and Match Functions	226
1.257jade.guide/Searching Buffers	227
1.258jade.guide/String Matching	228
1.259jade.guide/Replacing Strings	229
1.260jade.guide/Regexp Functions	229
1.261jade.guide/Rectangular Editing	231
1.262jade.guide/Controlling Undo	231
1.263jade.guide/Misc Text Functions	232

1.264jade.guide/Writing Modes	232
1.265jade.guide/Writing Major Modes	233
1.266jade.guide/Installing Modes	235
1.267jade.guide/Writing Minor Modes	236
1.268jade.guide/Mode-Specific Indentation	237
1.269jade.guide/Mode-Specific Expressions	238
1.270jade.guide/Mode-Specific Comments	239
1.271jade.guide/Prompting	240
1.272jade.guide/Files	242
1.273jade.guide/File Names	243
1.274jade.guide/File Objects	244
1.275jade.guide/Creating File Objects	245
1.276jade.guide/Destroying File Objects	245
1.277jade.guide/File Object Predicates	246
1.278jade.guide/Functions on File Objects	246
1.279jade.guide/File Information	246
1.280jade.guide/Manipulating Files	248
1.281jade.guide/Reading Directories	248
1.282jade.guide/Reading and Writing Files	248
1.283jade.guide/Processes	249
1.284jade.guide/Process Objects	249
1.285jade.guide/Asynchronous Processes	251
1.286jade.guide/Synchronous Processes	253
1.287jade.guide/Process I-O	254
1.288jade.guide/Process States	254
1.289jade.guide/Signalling Processes	256
1.290jade.guide/Process Information	256
1.291jade.guide/Interactive Processes	257
1.292jade.guide/Miscellaneous Functions	259
1.293jade.guide/System Information	259
1.294jade.guide/User Information	260
1.295jade.guide/Environment Variables	260
1.296jade.guide/System Time	261
1.297jade.guide/Revision Information	261
1.298jade.guide/Debugging	262
1.299jade.guide/Tips	263
1.300jade.guide/Comment Styles	264
1.301jade.guide/Program Layout	265
1.302jade.guide/General Tips	266

1.303jade.guide/Reporting Bugs	267
1.304jade.guide/Function Index	268
1.305jade.guide/Variable Index	295
1.306jade.guide/Key Index	301
1.307jade.guide/Concept Index	316

Chapter 1

jade

1.1 jade.guide

Jade

Jade is a highly flexible Emacs-style text editor for X11 (on Unix) and AmigaDOS.

This is Edition 1.3 of its documentation, last updated 7 October 1994 for Jade version 3.2.

Copying

Distribution conditions

Introduction

Brief introduction to Jade

News

New features in this release

Systems Supported

The operating systems Jade supports

Editor Concepts

Some ideas you should understand

Key Names

How keys are described in this manual

Starting Jade

How to start the editor

Using Jade

Instructions for using the editor

Programming Jade

How to extend Jade -- its Lisp system

Reporting Bugs

How to contact me

Function Index

Menu of all documented functions

Variable Index

All variables which have been mentioned

Key Index

Menu of all key bindings

Concept Index

Main index, references to all sections

1.2 jade.guide/Copying

Copying

Jade is distributed under the terms of the GNU General Public License, this basically means that you can give it to anyone for any price as long as full source code is included. For the actual legalese see the file 'COPYING' in the distribution. I reserve the right to use a different licence in future releases.

The only parts of Jade which are not my own work are the regexp code, this is by Henry Spencer (though I have made some small modifications) and is distributed under his conditions, and the ARexx interface in the Amiga version which is based on 'MinRexx' by Radical Eye Software.

Be aware that there is absolutely NO WARRANTY for this program, you use it at your own risk. Obviously I hope there are no bugs, but I make no promises regarding the reliability of this software.

1.3 jade.guide/Introduction

Introduction

Jade is a text editor primarily designed for programmers. It is easily customised through a Lisp-style extension language and can be tailored to the user's own requirements.

Jade is designed to run under a graphical windowing system, systems currently supported are the Commodore Amiga and the X Window System version 11 (but only under Unix).

It is the successor to the editor 'Jed 2.10' which I released for the Amiga in early 1993. I have decided to rename it now that I have made an X11 version since there is already an editor called 'Jed' available (there is no connection between the two, I haven't even looked at the other one). "Jade" is an anagram of "A Jed", if you want an acronym you could use "Just Another Damn Editor", if you can think of anything better please tell me.

Jade is compatible with GNU Emacs in terms of key presses and command names to a certain extent but it is not intended as a simple copy of Emacs (indeed, when I started this I had never actually used Emacs!). I have tried to take my favourite aspects of all the editors I have used as well as adding features that I have not found elsewhere. Consequently, it is very much the editor that *I* want -- you may not find it so appealing.

1.4 jade.guide/News

News

This chapter lists the major changes to Jade and which release they occurred in. Only changes relevant to you, the user, are detailed; for more explicit history see the 'ChangeLog' files with the sources.

Version 3.2

=====

- * The programmer's manual has finally be written.
 - * Undo; devote as much memory as you want to keep track of all modifications to a buffer which can then be wound back.
 - * Arguments can be given to commands as they're invoked.
 - * Buffer menu for interactive buffer manipulation.
 - * An Emacs-style local variables section can be embedded in a file; replaces the naff '::jade-code::' thing.
 - * 'Ctrl-k' ('kill-line') works at last.
 - * Now possible to interrupt jade while it's working (i.e. to let you kill infinite loops).
 - * The Help system now has commands to list key bindings, display what is bound to any key sequence.
 - * Use of the Latin-1 character set is now controlled by the minor mode 'latin-1-mode'.
-

- * Can load and save compressed (compress or gzip) files into/out of buffers transparently when running on Unix.
- * Transposing commands; 'transpose-chars', 'transpose-words', 'transpose-exps'. Bound to 'Ctrl-t', 'Meta-t' and 'Ctrl-Meta-t' respectively.
- * Can now run a shell in an editor buffer, very basic (no completion) but it works okay.
- * Support for using gdb through the shell interface, the current frame's source code is highlighted in a separate window.
- * 'Ctrl-z' moves to 'Ctrl-W' so that 'Ctrl-z' can (de)iconify the current window.
- * Some programs written for the previous incarnation will need to be altered; all will have to be recompiled.

Version 3.1

=====

- * Now properly supports characters which print as more than one character (i.e. proper tabs, '^L', '\123', etc...). In general any character can print as any sequence of up to four character-images.
 - * Doesn't expand tabs to spaces anymore, this means that loading and saving of largish files is noticeably quicker.
 - * Files containing NUL characters can be edited (more or less) successfully. Some commands (notably the regexp matcher) still don't like these characters but, in the main, binary files can be edited successfully.
 - * Searching and replacing has changed, it's easier to use now and replacing globally is built in.
 - * Many improvements to the Info viewer, not least, the dir file doesn't have to have a tag-table anymore.
 - * Client editing. This lets you load files into a running editor from a shell. For example, if your mailer runs an editor on the message you're writing you can use the client to edit the message in a Jade that you are running.
 - * The buffer prompt's completion is now controllable by the mouse as well as the keyboard. Click the right button to complete the current word. Double-clicking the left mouse button on one of the lines under the '::Completions::' line selects that completion.
 - * 'text-mode' and 'indented-text-mode' major-modes for editing English language (as opposed to programming languages).
 - * Minor-modes. These provide small variations to the major-modes. For example, 'overwrite-mode' makes typed keys overwrite whatever's under the cursor. Also included is a minor mode to do
-

auto-filling (word wrap).

- * On Unix, a tilde ('~') in a filename is handled properly in most cases
- * It is now possible to Meta qualify a key press and it will pretend that you pressed ESC then the un-Meta'd key.

1.5 jade.guide/Systems Supported

Requirements

Jade will only run on certain operating systems, this chapter details just what it needs as well as some notes relevant to each system.

Amiga Jade

=====

The only real requirement for Jade running on an Amiga is that it must run an operating system revision of at least V37 (thats V2.04) and have about 300K free memory available.

It also needs more stack than the average Amiga application. For normal use 20K should be okay. If you want to use the Lisp compiler 50K would be a better bet.

It assumes that its directory is pointed to by the 'JADE:' assignment. This means that the main Lisp files are stored in 'JADE:lisp/' and the file of doc-strings is 'JADE:DOC'.

X11 Jade

=====

Jade will only run on version 11 of X, it has absolutely no support for character terminals or different windowing systems. As long as it compiles it should work on your system.

One problem you might find is that the Backspace and Delete keys don't work properly. As far as I have been able to find out, most X terminals map both the Backspace (normally at the top-right of the alpha-keyboard) and the Delete (normally somewhere above the cursor keys) keys to the 'Delete' keysym. Obviously, since I want these keys to have different effects (1) this is no good. What I decided to do about this was two things,

1. Use 'xmodmap' to map the Delete key to the 'Backspace' keysym. This may sound backwards but most programs seem to use the 'Delete' keysym as what I call 'Backspace' so mapping as I described doesn't break this.

To do this, I have the following in my '.Xmodmap' file

keycode 107 = Backspace

Note that the '107' is the Delete key's key code on *my* keyboard, your keyboard may, and probably will, be different.

2. In the function which binds descriptions of key presses to Lisp forms, swap the meanings of the 'Backspace' and 'Delete' keysyms.

This means that everything works okay! You can bind to Delete key and it will work properly.

----- Footnotes -----

(1) Backspace should rub out the key before the cursor and Delete should delete the character under the cursor

1.6 jade.guide/Editor Concepts

Editor Concepts

Before I describe the editor in detail there are several concepts which you should be familiar with. Some will be explained in more detail later.

"buffer"

Buffers are used by the editor to store the text that you are editing. Broadly speaking, each buffer holds the contents of one text-file loaded into the editor (it is not necessary for each buffer to be associated with a file, some buffers exist for other purposes for example the '*jade*' buffer is used to interact with the Lisp system).

"current buffer"

The buffer being edited in the current window (see below), most editor commands work on this buffer unless told otherwise.

"window"

Corresponds to a window in the window-system. Each window can display one buffer at a single time (although a buffer may be displayed in more than one window at once).

"current window"

Jade always keeps track of which one of its windows is active. It is called the current window. Whenever you type a key or press a mouse button in one of Jade's windows, that window automatically becomes the current window. Amongst other things, all messages from the editor are displayed in the status line of the current window.

"cursor"

The cursor marks your current position in the current buffer (see above), when you type something it is inserted into the buffer between the cursor and the character preceding it (unless you type

a command).

"status line"

One line in a window is devoted to displaying messages from the editor,

Using Windows

.

"Lisp"

The programming language which Jade uses, although the internals of the editor are written in C, all commands are written in a dialect of Lisp (even if the command only calls a C function). Jade contains an interpreter, compiler and debugger for this language. See

Programming Jade

.

"variable"

Variables are used to store Lisp values, each variable has a unique name. Note that unlike many programming languages variables in Lisp are **not** typed, the data values themselves have a type associated with them.

"form"

A form is a single Lisp expression. For example, all of these are forms:

```
foo
42
"hello"
(setq foo 200)
```

"command"

A command is a sequence of Lisp forms which may be called interactively (i.e. from the keyboard). It may be a key sequence (such as `'Ctrl-x Ctrl-f'`) or a Lisp function to evaluate (such as `'find-file'`).

"regular expression"

A regular expression is a string which is used to match against other strings. It has a special syntax which allows you to form a kind of template against which the other strings can be matched. They are used extensively by the editor, but you -- the user -- will mainly encounter them when searching and replacing strings in buffers.

1.7 jade.guide/Key Names

Key Names

In this manual I have adopted a consistent notation for all key presses, since most editor commands are invoked via a typed key

sequence it is very important that you can decipher this notation.

Note that the term 'input event' (or 'event') and the term 'key press' have been used interchangeably throughout this manual. A 'key press' may mean a mouse event, they don't always come from the keyboard.

Every key press has a set of "modifiers"; these are the keys such as "Shift" or "Control" which don't actually produce a character when typed, they only effect the rest of the keyboard. Each key, then, can have one or more modifiers.

The name of an event consists of zero or more hyphen-separated modifier names, followed by a hyphen and the name of the actual event.

Some commands are triggered by more than one of these key presses; press each key (or do whatever is necessary to precipitate the input event) in turn to invoke the command.

Note that the case of modifiers is not important, however some of the keys *are*, so you should always specify them in their correct case.

Modifiers	Names of modifier keys
Keys	Names of actual keys
Example Keys	Some examples and what they mean

1.8 jade.guide/Modifiers

Modifiers
=====

"Shift"

"SFT"

The shift key.

"Ctrl"

"CTL"

The control key, or its equivalent.

"Meta"

This depends on the window-system, on X11 it is the "Mod1" modifier, on the Amiga the "Alt" key. When the 'meta-sends-esc' variable is non-nil the Meta modifier is treated specially,

- Variable: meta-sends-esc

When non-nil, any Meta-modified key presses are expanded into a sequence of two key presses, ESC and the pressed key minus

its Meta modifier. For example typing `'Meta-f'` would expand to `'ESC f'`. This feature is provided for compatibility with GNU Emacs.

What this really means is that when the option is enabled (it is by default) you can either type the key sequence `'ESC X'` or the sequence `'Meta-X'` (where Meta is your keyboard's meta key) to invoke a command described as `'Meta-X'`.

"LMB"

The left mouse button.

"MMB"

The middle mouse button.

"RMB"

The right mouse button.

As well as these, there are also some others, "Mod1" to "Mod5" represent the X11 modifiers of the same name. "Button1" to "Button5" also correspond to their X11 counterparts (Button1 to Button3 are LMB to RMB). For Amiga users, "Amiga" corresponds to the Amiga key (this is the same as Mod2).

1.9 jade.guide/Keys

Keys

====

As far as possible each single character key-definition corresponds to where that character is on the keyboard (a is `'a'`, etc...).

When using an Amiga this should be true for **all** keys since the Amiga's "keymap.library" makes it easy to look up what key a character belongs to. However, this is not so easy on X11. All of the standard ASCII character set should be okay, but the more esoteric characters may have to be specified by the names of their X11 keysym (without the `'XK_'` prefix). Look in the `<X11/keysymdef.h>` include file for all keysyms, for example `'XK_question'` would have to be used for `'?'` if the editor didn't treat it, and many others, specially.

Some keys which don't follow this pattern are

"SPC"

"Space"

The space bar.

"TAB"

The tab key.

"RET"

"Return"

The return key.

"ESC"
"Escape"
 The escape key.

"BS"
"Backspace"
 The backspace key.

"DEL"
"Delete"
 The delete key.

"Help"
 The help key, not all keyboards have this.

"Up"
 The cursor up key.

"Down"
 The cursor down key

"Left"
 The cursor left key.

"Right"
 The cursor right key.

"KP_Enter"
"KP_Multiply"
"KP_Divide"
"KP_Minus"
"KP_Add"
"KP_Decimal"
"KP_N"
 Keys on the numeric keypad. For KP_N, N is a digit.

"Click1"
 Single clicking a mouse button.

"Click2"
 Double clicking a mouse button.

"Off"
 Releasing a mouse button.

"Move"
 Moving the mouse. This doesn't work on X11 yet.

1.10 jade.guide/Example Keys

Example Keys

=====

Some examples of proper key names are,

``Ctrl-x``

Hold down Control, type x.

``Meta-Shift-RET``

Hold down Meta and Shift, then type the Return key, or alternatively, type the Escape key then hold down Shift and type Return.

``LMB-Click1``

Click the left mouse button once.

``Ctrl-RMB-Click1``

Hold down Ctrl then click the right mouse button once.

1.11 jade.guide/Starting Jade

Starting Jade

This chapter describes Jade's initialisation process. This includes how to start it, what options it will accept and what it actually does after being started.

Invocation

How to start the editor

Startup Options

Arguments specified on the command line

Startup Procedure

What happens on startup

1.12 jade.guide/Invocation

Invocation

=====

Since Jade supports two vastly different operating systems they both need to be covered separately.

Amiga

The normal way to start Jade on the Amiga is to type its name at the Shell (or CLI) together with any options (see

Startup Options

) you

want. Note that these options are in the traditional Unix style, a dash followed by the option name and any arguments, not the standard AmigaDOS method.

It is also possible to invoke the editor from the Workbench, simply double clicking on its icon will cause Jade to open its initial window. Unfortunately there is no support for passing arguments via Tool Types, nor is there any way to create icons with saved files. This is largely due to the fact that I rarely use the Workbench -- if enough people complain about this I will probably fix it. Jade doesn't have an icon yet, you'll have to make one yourself.

X11

Jade should be started like most other Unix programs, type its name and any arguments to a shell. It must be able to connect to an X server (preferably the one controlling your terminal), the '-display' option can be used if needed.

1.13 jade.guide/Startup Options

Startup Options

=====

The acceptable options can be split into three classes. Note that they must be specified on the command line in order of their class. This means that, for example, the '-rc' option must be after the '-font' option.

So, the general usage pattern is

```
jade [SYSTEM-DEPENDENT-OPTIONS] [STANDARD-OPTIONS] [LISP-OPTIONS]
```

Note that the LISP-OPTIONS may include files to be loaded.

1. System dependent options.

* Options for the Amiga system.

'-pubscreen SCREEN-NAME'

Defines the name of the public screen on which the first window is opened. By default (or if SCREEN-NAME doesn't exist) the 'Workbench' screen is used.

'-font FONT-STRING'

Defines the font used in the first window. FONT-STRING is the font to use, it is the name of the font (for example, 'topaz.font'), followed by a hyphen and the point size to use. For example, a FONT-STRING of 'topaz.font-8' gives 8-point topaz. This is the default.

`'-stack STACK-SIZE'`

When this argument is given Jade allocates a new stack. STACK-SIZE is a decimal number defining the size (in bytes) of the new stack.

If this argument is not given Jade simply uses the stack that AmigaDOS gave it.

* Options for X11.

There are two types of options to the X11 version of the editor, those specified on the command line and those defined in the resource database (i.e. in your `.Xdefaults` file). Resources are looked for under two names, firstly the name under which the editor was invoked (normally `'jade'`), if this fails it tries again with the name `'Jade'`. Naturally, options specified on the command line override those in the resource database.

`'-display DISPLAY-NAME'`

Defines the name of the X display to open, by default the contents of the environment variable `'DISPLAY'`. It is a string of the form `'HOST-NAME:NUMBER.SCREEN-NUMBER'`.

`'-name NAME'`

The name to use when looking up resource values, this replaces the base name of the executable (normally `'jade'`).

`'-geometry GEOM-SPEC'`

Specifies where to place the first window on the screen. This is a standard X style geometry specification.

`'-fg FOREGROUND-COLOUR'`

Resource: `'fg: FOREGROUND-COLOUR'`

The colour of the window's foreground (i.e. the text).

`'-bg BACKGROUND-COLOUR'`

Resource: `'bg: BACKGROUND-COLOUR'`

The background colour of the window.

`'-font FONT-NAME'`

Resource: `'font: FONT-NAME'`

The name of the font used for all text in the initial window.

2. Standard options.

`'-rc LISP-FILE'`

Load the Lisp script LISP-FILE instead of the normal initialisation script (`'init'`). Warning: the editor depends heavily on the normal file, if you change this without due care the editor could be unusable -- no keys will be bound and many standard functions won't exist.

`'-v'`

Print the version and revision numbers of this copy of the

editor then quit.

`'-log-msgs'`

This option makes all messages which are displayed in the status line also be written to the standard error stream. This is sometimes useful for debugging purposes.

3. All other options are passed to the Lisp initialisation process in the variable `'command-line-args'`, these are available to any Lisp packages loaded in the initialisation script. Any left after that are scanned for the following options,

`'-f FUNCTION'`

Call the Lisp function `FUNCTION`.

`'-l FILE'`

Load the Lisp file `FILE`.

`'-q'`

Quit cleanly.

`'FILE'`

Load the file of text `FILE` into a new buffer.

An example command line for starting Jade from a Unix shell could be

```
$ jade -fg white -bg black -log-msgs foo.c bar.jl
```

This means white text, black background, save messages and load the files `'foo.c'` and `'bar.jl'`.

1.14 jade.guide/Startup Procedure

Startup Procedure

=====

This is a description of what happens when the editor initialises itself.

1. Firstly lots of internal data structures are created, memory pools, symbols and their symbol-table (including all the primitive Lisp functions).
 2. The window-system is initialised (parse the system-dependent options, and the `xrdb` resources if in X).
 3. Parse the standard options.
 4. Create the initial window and the first buffer to display in it (this is the buffer called `'*jade*'`).
 5. Load the initialisation script, this is either the Lisp file called `'init'` or whatever was given to the `'-rc'` command line option.
-

Some selected highlights of what the standard file does are,

* Load lots of Lisp files, some notable ones are

 `autoload`

 Initialise the autoload stubs.

 `loadkeys`

 Creates the standard keymaps and key bindings.

* Try to find the user's personal startup file, this is normally the file `.jadarc` in their home directory (1).

* Load any files which were specified on the command line.

6. Start the top-level recursive edit, this doesn't exit until the editor does.

----- Footnotes -----

(1) The Amiga has no notion of a user's home directory, Jade uses the contents of the environment variable `HOME`, or if this doesn't exist the `SYS:` assignment.

1.15 jade.guide/Using Jade

Using Jade

This chapter of the manual is meant to teach you to *use* the editor, because of this I have attempted to reduce references to the Lisp extension language to an absolute minimum.

Invoking Commands

How to use the commands and key-sequences described in this manual.

Command Arguments

Many commands can be modified by prefixing them with a numeric argument

The Help System

Online help facilities

Loading and Saving Files

Manipulating files

Editing Buffers

Simple editing commands

Moving Around Buffers
 Commands for moving the cursor

Undo
 Go back in time

Editing Units
 Characters, words, lines, etc...

Cutting And Pasting
 How to insert text from the clipboard

Using Blocks
 Highlighting regions to manipulate

Killing
 Deleting text for later insertion

Searching and Replacing
 Searching the buffer for a regexp

Editing Modes
 Editing different types of files

Minor Modes
 Small alterations to editing modes

Using Buffers
 Selecting & deleting buffers

Using Windows
 Opening new windows

Using the Prompt
 Entering strings and completion

Using Marks
 Recording positions in files

Interrupting Jade
 Breaking out of commands

Recursive Editing
 Editing within a command

Character Images
 How to get a Latin1 character set

Client Editing
 Using Jade from other programs

Compiling Programs

Help for developing programs

Info Mode

Reading Info files with Jade

Shell

Using a shell inside a buffer

Simple Customisation

Configuring Jade

1.16 jade.guide/Invoking Commands

Invoking Commands

=====

Throughout this manual I have documented the key sequences you have to enter to make the editor perform a certain action. In fact, the key sequences are mapped to "commands" when they are typed and it is the *command* which performs the action.

Commands are simply pieces of Lisp code, usually with a unique name associated with that command. If you know the name of a command it is possible to invoke it using the 'Meta-x' key sequence; simply type 'Meta-x COMMAND RET' where COMMAND is the name of the command you wish to invoke.

'Meta-x'

Prompt for the name of a command (completion is available) then invoke it.

For the sake of simplicity I have often referred to key sequences as commands; what I actually mean is that the key sequence is bound to the command. For example the key sequence 'Ctrl-x Ctrl-f' opens a file, in fact the key sequence 'Ctrl-x Ctrl-f' is bound to the command 'find-file', this Lisp function actually loads the file.

More detailed information about commands is available in the programmer's manual, see

Programming Jade

.

1.17 jade.guide/Command Arguments

Command Arguments

=====

The actions of many commands can be altered by giving them a numeric

argument, this argument is entered immediately prior to invoking the command (they are technically called prefix arguments).

Each argument is built using a number of special key sequences,

`'Meta-0'` to `'Meta-9'`

Append a digit to the end of the current prefix argument. Use a sequence of these keys to build up a decimal number. For example typing `'Meta-1 Meta-0 Meta-0'` creates an argument of 100 for the following command.

`'Meta--'`

(That's `'Meta-minus'`.) Negates the value of current argument, if the command is invoked after a single `'Meta--'` prefix the actual argument is `-1`.

`'Ctrl-u'`

Successive `'Ctrl-u'` key presses multiply the argument by 4 each time. Note that any existing argument entered by the numeric or minus commands (described above) is discarded with the first `'Ctrl-u'`.

1.18 jade.guide/The Help System

The Help System

=====

To invoke the help system type the key sequence `'Ctrl-h'` or if your keyboard has it the `'HELP'` key.

A prompt will be displayed in the status line showing you which keys you can press next to enter one of the main options of the help system explained below. Alternatively, you can type either `'Ctrl-h'` or `'HELP'` again to display some text telling you more about the help system and how to use it.

The help system is exited after successfully invoking one of the commands described below or typing anything which is not a recognised command to the help system.

`'a'`

To list all function names matching REGEXP, type `'a REGEXP RET'` when in the help system.

`'b'`

Prints all key bindings and their associated commands which are installed in the current buffer.

`'e'`

Similarly to the `'a'` command, to list all variable names matching REGEXP, type `'e REGEXP RET'` when in the help system.

`'f'`

Displays the online documentation for a function. After invoking

this option type the name of the function.

`'h'`

Shows some helpful text describing how to use the help system.

`'i'`

Enters the Info viewer. This allows you to browse through files written in the Info hypertext format. For more information see

Info Mode

, for more information on Info files in general see Info.

`'k'`

Displays the command (with its documentation) for a key sequence. After typing `'Ctrl-h k'` enter the key sequence you want documented as if you were going to invoke the command.

`'m'`

Display the documentation for the current major mode.

`'v'`

Displays the online documentation and current value of a variable. Type the name of the variable after invoking this option.

1.19 jade.guide/Loading and Saving Files

Loading and Saving Files

=====

Since `'Jade'` is a text editor its main function is to edit files of text. This means that you must be able to read the text contained in a file into one of the editor's buffers, then save it back to disk when you have finished editing it. That is what this section deals with.

Commands To Load Files

Key sequences to load files

Commands To Save Files

How to save a buffer

File Variables

Local variables defined in files

Backup Files

Making backups

Auto-Saving Files

Files can be saved periodically

Accessing Compressed Files

Reading and writing gzipped files

Other File Commands

How to delete, rename or copy files

1.20 jade.guide/Commands To Load Files

Commands To Load Files

There are several commands used to load files into buffers, these are,

``Ctrl-x Ctrl-f'`

Prompts for the name of a file (using file-completion) and display the buffer containing that file. If the file has not already been loaded it will be read into a new buffer.

``Ctrl-x Ctrl-v'`

Prompts for the name of a file, the current buffer is killed and the buffer in which the prompted-for file is being edited is displayed. As in `'find-file'` it will be read into a new buffer if it is not already in memory.

``Ctrl-x Ctrl-r'`

Similar to `'find-file'` except that the buffer is marked as being read-only. This means that no modifications can be made to the buffer.

``Ctrl-x i'`

Prompts for a file, then inserts it into the current buffer at the cursor position.

You can use the prompt's completion feature to expand abbreviated filenames typed to the prompt, for more information see

The Buffer Prompt

.

1.21 jade.guide/Commands To Save Files

Commands To Save Files

These are the commands used to save buffers and the key sequences associated with them,

``Ctrl-x Ctrl-s'`

Saves the current buffer to the file that it is associated with

(this is either the file that it was loaded from or something else set by the function `'set-file-name'`). If no modifications have been made to the file since it was loaded it won't be saved (a message will be displayed warning you of this).

`'Ctrl-x Ctrl-w'`

Prompts for a name to save the file as. The file associated with this buffer is renamed and the file is saved as its new name.

`'Ctrl-x s'`

For each buffer which has been modified since it was loaded, ask the user if it should be saved or not. If so, the command `'save-file'` is used to save the file

1.22 jade.guide/File Variables

File Variables

It is often useful to define `'local'` values of certain variables which only come into effect when a particular file is being edited. Jade allows you to include a special section in a file which, when the file is loaded, is used to give the variables specified buffer-local values. (For more information about buffer-local variables see

Buffer-Local Variables

.)

The special section must be somewhere in the last twenty lines of a file, and must be formatted as in the following example,

```
XXX Local Variables: YYY
XXX VARIABLE:VALUE YYY
...
XXX End: YYY
```

That is, the string `'Local Variables:'` followed by as many lines defining local values as necessary then the string `'End:'`. The two strings `'XXX'` and `'YYY'` may be anything (even nothing!) as long as they are the same on each line. They are normally used to put the local variable section into a comment in a source file.

For example, in a Texinfo source file the following piece of text at the bottom of the file would set the column at which lines are broken to 74 (note that `'@c'` introduces a comment in Texinfo).

```
@c Local Variables:
@c fill-column:74
@c End:
```

Two pseudo-variables which can be set using a local variables section are `'mode'` and `'eval'`. Setting the `'mode'` variable actually defines the major mode to use with the file (see

Editing Modes

) while setting
'eval' actually evaluates the Lisp form VALUE then discards its value.

For example,

```
/* Local Variables: */  
/* mode:C */  
/* eval:(message "How pointless!") */  
/* End: */
```

This Forces the file to be edited with the C mode and displays a pointless message. Note that no variables called 'mode' or 'eval' are actually set.

Several variables are used to control how the local variables feature works.

- Variable: enable-local-variables
Defines how to process the 'Local Variables:' section of a file: 'nil' means to ignore it, 't' means process it as normal and anything else means that each variable being set has to be confirmed by the user. Its default value is 't'.
- Variable: enable-local-eval
This variable defines how the pseudo-variable 'eval' is treated in a local variables list, it works in the same way as the 'enable-local-variables' variable does. Its default value is 'maybe', making each form be confirmed before being evaluated.
- Variable: local-variable-lines
Defines how many lines at the bottom of a file are scanned for the 'Local Variables:' marker, by default it is 20.

Note that this feature is compatible with GNU Emacs, and since I have tried to keep the names of variables compatible as well, there should be few problems.

1.23 jade.guide/Backup Files

Backup Files

The editor can optionally preserve the previous contents of a file when it is about to be overwritten by the saving of a buffer. It does this by renaming the old file, 'foo' as 'foo~' (the original name plus a tilde appended to it) before it is obliterated.

- Variable: make-backup-files
This variable controls whether or not backups are made of files about to be overwritten by the function 'write-buffer' (i.e. the commands 'save-file' and 'save-file-as'). When non-nil the old instance of the file is renamed so that it has a tilde appended to its old name.

- Variable: backup-by-copying
When non-nil all backups are made by copying the original file instead of renaming it as the backup file. This is slower but less destructive.
- Variable: else-backup-by-copying
If 'backup-by-copying' is 'nil' and renaming the original file would not be a good idea (i.e. it might break a link or something) and this variable is non-'nil' the backup will be made by copying the original file.

1.24 jade.guide/Auto-Saving Files

Auto-Saving Files

Jade is able to save snapshots of a buffer's contents at set time intervals. When this time interval expires and the buffer has been modified since it was last (auto-) saved to disk (and the editor is idle) the buffer is saved to a special file (usually the base component of the file's name surrounded by '#' characters in the file's directory).

- Variable: auto-save-p
When non-nil this makes the function 'open-file' (and therefore the commands 'find-file', etc) flag that the file it just read should be auto saved regularly.
- Variable: default-auto-save-interval
This is the default number of seconds between each auto save. This variable is only referenced when each file is opened.

Its standard value is 120 seconds.
- Variable: auto-save-interval
This buffer-local variable controls the number of seconds between each auto-save of the buffer it belongs to. A value of zero means never auto-save.

When the buffer is saved properly (i.e. with 'save-file' and friends) its auto-save file is deleted. Note that this doesn't happen when you kill a buffer and an auto-save file exists (in case you didn't mean to kill the buffer).

To recover an auto-saved file (i.e. after an editor crash or something!) use the command 'recover-file'.

'Meta-x recover-file'

Loads the auto-saved copy of the file stored in this buffer overwriting its current contents (if any changes are to be lost you will have to agree to losing them).

1.25 jade.guide/Accessing Compressed Files

Accessing Compressed Files

Jade contains basic support for reading, inserting and writing buffers which have been compressed using the 'gzip' or 'compress' compression programs. When this feature is enabled such files are transparently decompressed when loaded into the buffer and compressed when saved back to a file.

Unfortunately this doesn't work on Amigas yet. To install it the Lisp form,

```
(require 'gzip)
```

should be in your '.jadarc' file (or you can do this by hand in the '*jade*' buffer if you want).

After the 'gzip' package has been installed any files loaded into buffers whose filename end in '.gz' or '.Z' are uncompressed, this suffix is stripped when searching for a major mode to install in the buffer but otherwise the buffer's filename is left intact.

Any buffer saved whose filename ends in one of the above suffixes is automatically compressed ('.gz' is compressed by 'gzip', '.Z' by 'compress').

1.26 jade.guide/Other File Commands

Other File Commands

```
`Meta-x delete-file RET FILE-NAME RET'  
  Deletes the file called FILE-NAME.
```

```
`Meta-x rename-file RET SOURCE RET DEST RET'  
  Renames the file called SOURCE as the file DEST.
```

```
`Meta-x copy-file RET SOURCE RET DEST RET'  
  Makes a copy of the file called SOURCE as the file DEST.
```

1.27 jade.guide/Editing Buffers

Editing Buffers

=====

The majority of keys when typed will simply insert themselves into the buffer (this is not always true but it's a good assumption) since they have not been bound. Typically this includes all normal characters

(i.e. alphanumeric, punctuation, etc) as well as any of the more obtuse key-sequences which have not been bound to a function ('Ctrl-l' is one of the more useful of these).

The behaviour of the TAB key is different to many other editors -- it doesn't insert anything (unless a specific editing mode has bound it to something else, like 'c-mode' for example), generally it just moves the cursor to the next tab stop. This is partly because Jade doesn't use "proper" tabs and partly because it makes it easier to move around a line (because the key sequence 'Shift-TAB' moves to the previous tab stop).

Some miscellaneous editing commands follow.

'RET'

This generally splits the line into two at the position of the cursor, some editing modes may provide an option which automatically indents the line after it's split.

'Backspace'

Deletes the character before the cursor.

'DEL'

'Ctrl-d'

Deletes the character under the cursor.

'Shift-Backspace'

Kills the characters between the start of the line and the cursor.
See

Killing

.

'Shift-DEL'

Kills the characters from the cursor to the end of the line.

'Ctrl-DEL'

Kills the whole line.

'Ctrl-o'

Splits the line in two at the cursor, but leaves the cursor in its original position.

'Meta-d'

'Meta-DEL'

Kills from the cursor to the end of the current word.

'Ctrl-k'

Kills from the cursor to the end of the line, or if the cursor is at the end of the line from the cursor to the start of the next line. Each successive 'Ctrl-k' appends to the text in the kill buffer.

'Meta-l'

Makes the characters from the cursor to the end of the word lower case.

'Meta-u'

Upper cases the characters from the cursor to the end of the word.

``Meta-c'`

Capitalises the characters from the cursor to the end of the word, this means make the first character upper case and the rest lower.

``Meta-Backspace'`

Kills from the cursor to the beginning of the word.

1.28 jade.guide/Moving Around Buffers

Moving Around Buffers

=====

Here is a selection of the most commonly used commands which move the cursor around the current buffer.

``Up'`

``Ctrl-p'`

Move one line up.

``Down'`

``Ctrl-n'`

Move one line down.

``Left'`

Move one column to the left, stopping at the first column.

``Ctrl-b'`

Move to the previous character, at the beginning of the line moves to the end of the previous line.

``Right'`

Move one column to the right. This keeps moving past the end of the line.

``Ctrl-f'`

Move to the next character, at the end of a line moves to the start of the next line.

``Shift-Up'`

Move to the first line in the buffer.

``Shift-Down'`

Move to the last line in the buffer.

``Meta-<'`

Move to the first character in the buffer.

``Meta->'`

Move to the last character in the buffer.

``Shift-Left'`

``Ctrl-a'`
Move to the beginning of the current line.

``Shift-Right'`
``Ctrl-e'`
Move to the last character in the current line.

``Ctrl-Up'`
``Meta-v'`
Move to the previous screen of text.

``Ctrl-Down'`
``Ctrl-v'`
Move to the next screen of text.

``Meta-Left'`
``Meta-b'`
Move to the previous word.

``Meta-Right'`
``Meta-f'`
Move to the next word.

``Meta-Up'`
``Meta-['`
Move to the start of the previous paragraph.

``Meta-Down'`
``Meta-]'`
Move to the start of the next paragraph.

``TAB'`
``Meta-i'`
Insert a tab character, indenting the cursor to the next tab position.

Note that some editing modes redefine TAB to make it indent the current line to its correct depth.

``Shift-TAB'`
Move to the position of the previous tab.

``Ctrl-TAB'`
Move to the position of the next tab.

``Meta-j'`
Prompt for a line number and go to it.

``Meta-m'`
Move to the first non-space character in the current line.

1.29 jade.guide/Undo

Undo

=====

Jade makes it very easy to undo changes to a buffer, this is very useful when you realise that actually, **that wasn't** the part of the file you wanted to delete!

Basically to undo the last command type either `'Ctrl-_'` or `'Ctrl-x u'`. If the last thing you did was to type some text into the buffer all the consecutively-typed characters count as one command.

To undo more than one command, simply type more than one `'Ctrl-_'` (or `'Ctrl-x u'`) consecutively; this will progressively work its way back through the buffer's history. The first non-undo command cancels this effect, so if you undo too far back invoke a command which doesn't modify the buffer, then undo whatever you undid.

`'Ctrl-_'`

`'Ctrl-x u'`

Undo the previous command, or the last block of consecutively typed characters. Successive undo commands work backwards through the undo-history until a non-undo command is invoked.

The exact amount of undo-information kept for each buffer is controlled by the `'max-undo-size'` variable. This defines the maximum number of bytes which may be devoted to undo-information in a single buffer, the default is 10000. No matter what this is set to, the last command is **always** recoverable.

- Variable: `max-undo-size`

The maximum memory which may be devoted to recording undo-information in each buffer.

1.30 jade.guide/Editing Units

Editing Units

=====

To make it easier to remember which key sequences do what Jade provides a number of commands which are similar to one another but operate on different "units" in the buffer. These related-commands are bound to the same key but with a different prefix or modifier. For example `'Ctrl-f'` moves forward one character while `'Meta-f'` moves forward one word.

Editing Characters

Commands operating on characters,

Editing Words

words,

Editing Expressions

expressions,

Editing Lines

and lines.

1.31 jade.guide/Editing Characters

Editing Characters

These are the commands which operate on characters. Note that when an argument (see

Command Arguments

) is given to one of these commands it

actually operates on **number** of characters. For example, if you want to delete the next 5 characters starting at the cursor type `'Meta-5 Ctrl-d'`.

`'Ctrl-f'`

Move forward one character.

`'Ctrl-b'`

Move back one character.

`'Right'`

Move one character to the right, when the end of the line is encountered it's ignored and the cursor keeps moving to the right.

`'Left'`

Move one character to the left, stops when the beginning of the line is reached.

`'Ctrl-d'`

`'DEL'`

Deletes the character beneath the cursor.

`'Backspace'`

Deletes the character before the cursor.

`'Ctrl-t'`

Transposes the character before the cursor with the one under the cursor. When given an argument the character before the cursor is dragged forward over that many characters.

`'Meta-SPC'`

Delete all white space characters surrounding the cursor leaving a single space in their place. If a prefix argument is given that many spaces are left.

`'Meta-\'`

Delete all white space characters surrounding the cursor. This is equivalent to the key sequence `'Meta-0 Meta-SPC'`.

1.32 jade.guide/Editing Words

Editing Words

The following commands operate on words. When given a prefix argument they operate on that number of words all in one go.

The syntax of a word depends largely on the major mode being used to edit the buffer with, see
Editing Modes

.

`'Meta-f'`

`'Meta-Right'`

Move forward one word.

`'Meta-b'`

`'Meta-Left'`

Move back one word.

`'Meta-d'`

`'Meta-DEL'`

Kills characters from the cursor to the start of the next word.

See

Killing

.

`'Meta-Backspace'`

Kills characters from the start of the previous word to the cursor position.

`'Meta-t'`

Transpose words: the word before the cursor is dragged over the following word. An argument means to drag the word over that number of words.

`'Meta-u'`

Convert the characters from the cursor to the start of the next word to upper-case.

`'Meta-l'`

Similar to `'Meta-u'` but converts to lower-case.

`'Meta-c'`

Capitalise the word beginning at the cursor position. What happens is that the next alphabetic character is converted to upper-case then the rest of the word is converted to lower-case. Note that an argument to this command currently has no effect.

1.33 jade.guide/Editing Expressions

Editing Expressions

Expressions are used when editing programming languages; the editing mode for a particular programming language defines the syntax of an expression element in that language. In other editing modes an expression is defined as a single word.

These commands use prefix arguments in the normal manner.

``Ctrl-Meta-f'`

Move forward over one expression element.

``Ctrl-Meta-b'`

Move backwards over one expression.

``Ctrl-Meta-k'`

Kills the following expression, starting from the current cursor position. A negative argument means kill backwards. See Killing

.

``Ctrl-Meta-t'`

Transpose the previous expression with the following one. An argument means to drag the previous one over that many expressions.

1.34 jade.guide/Editing Lines

Editing Lines

These commands all operate on one or more lines of text. Most use a prefix argument (if entered) to define how many lines to move or operate on.

``Ctrl-n'`

``Down'`

Move down one line.

``Ctrl-p'`

``Up'`

Move to the previous line.

``Ctrl-a'`

``Shift-Left'`

Move to the beginning of the current line.

``Ctrl-e'`

``Shift-Right'`

Move to the end of the current line.

``Meta-j'`

Prompts for the number of a line to jump to. If a prefix argument was entered that defines the line number.

``Ctrl-DEL'`

Kill the current line. See
Killing

.

``Shift-DEL'`

Kill from the cursor to the end of the current line.

``Shift-Backspace'`

Kill from the cursor to the beginning of the line.

``Ctrl-k'`

If the cursor is not at the end of the line kill the text from the cursor to the end of the line, else kill from the end of the line to the start of the next line.

If this command is given an argument it kills that number of *whole* lines, either backwards or forwards from the cursor, depending on whether or not the argument is negative or positive. An argument of zero kills from the cursor to the start of the current line.

``Ctrl-o'`

Create a blank new line, leaving the cursor in its original position. A prefix argument says to create that many blank lines.

1.35 jade.guide/Cutting And Pasting

Cutting And Pasting

=====

One of the main functions of any editor is to allow you to move around chunks of text, Jade makes this very easy.

Generally, to paste down some text you have to get the text to be inserted into the window-system's clipboard (1). If the text you wish to paste is in one of the editor's buffers Jade has a number of commands for doing this, this is sometimes referred to as "killing" the text. For details of how to kill a piece of text see

Killing

.

If the text to be pasted is in the same buffer as the position to which you want to copy it there is an easier way than putting it into the clipboard. For more details see

Commands on Blocks

and the command

``Ctrl-i'`.

Once the text to be pasted is in the clipboard there are two commands which can be used to insert it into the buffer before the cursor,

``Ctrl-y'`

Inserts text into the buffer before the cursor. The text inserted is either the current contents of the kill buffer, or the block marked in this window, if one exists.

``Ctrl-Y'`

This is a variant of ``Ctrl-y'`, it treats the string that it is pasting as a "rectangle" of text. That is, each successive line in the string (each separated by a newline character) is inserted on successive lines in the buffer but at the same column position.

For more details see

 Rectangular Blocks

 and the function

``insert-rect'`.

----- Footnotes -----

(1) When using an Amiga, unit zero of the ``clipboard.device'` is used. For X11, the first cut-buffer.

1.36 jade.guide/Using Blocks

Using Blocks

=====

A "block" is a section of a buffer, you mark it by specifying its edges (i.e. the first and last characters). This part of the buffer can then have various things done to it, for example insert it somewhere else.

Each window can only have a single block marked at any one time, it will be displayed in the reverse of normal text (i.e. white on black, not black on white).

Marking Blocks

 Commands to define the current block

Commands on Blocks

 How to work with blocks

Rectangular Blocks

 Columns of text as blocks

1.37 jade.guide/Marking Blocks

Marking Blocks

To mark a block you must specify its outermost points, note that the text marked by the block ends one character before the marked position (this is so that it is easy to mark whole lines).

Rectangular blocks are a bit different for more information, see

Rectangular Blocks

Note also that block marks shrink and grow as text is deleted and inserted inside them, similar to what normal marks do.

These are the commands used to mark a block,

`'Ctrl-m'`

`'Ctrl-SPC'`

If a block is currently marked in this window it will unmark it. Otherwise it will either mark the beginning or end of the block depending on whether or not a block has previously been partially marked.

The normal method for marking a few characters is to first make sure that no block is currently marked (the status line displays the status of the block marks, a 'b' means that one end of a block has been marked and a 'B' means that both ends of a block are marked in which case it will be highlighted somewhere in the buffer) then press `'Ctrl-m'` at one end, move the cursor to the opposite end and press `'Ctrl-m'` again.

`'Ctrl-x h'`

Mark the whole of the buffer.

`'Meta-@'`

Mark the current word.

`'Meta-h'`

Mark the current paragraph.

Another method for marking a block is to use the mouse, double clicking the left mouse button on a character has the same effect as moving to that character and typing `'Ctrl-m'`. Similarly, clicking the left mouse button while pressing the SHIFT key clears a marked block.

1.38 jade.guide/Commands on Blocks

Commands on Blocks

``Ctrl-i'`

Inserts the block marked in this window, at the cursor position, then unmarks the block.

``Ctrl-w'`

Kills the contents of the marked block, for information about killing see

Killing

.

``Meta-w'`

Similar to ``Ctrl-w'` except that the text is not actually deleted, just stored for later recall.

``Ctrl-W'`

Deletes the text in the currently marked block.

``Ctrl-x Ctrl-l'`

Makes all alpha characters in the current block lower case.

``Ctrl-x Ctrl-u'`

Makes all characters in the block upper case.

1.39 jade.guide/Rectangular Blocks

Rectangular Blocks

Normally blocks are thought of sequentially from their first to last characters. It is also possible to mark rectangular blocks, the block marks being thought of as the opposite corners of the rectangle.

The commands which operate on blocks automatically check whether the current block is a rectangle; if so they change their function accordingly. For example, the ``Ctrl-i'` command (``insert-block'`) understands that rectangular blocks have to be inserted in a different manner to normal, sequential, blocks.

``Ctrl-M'`

Toggle between marking sequential and rectangular blocks, each window has its own value of this attribute (i.e. one window can be marking rectangles while the rest don't).

``Ctrl-Y'`

Similar to ``Ctrl-y'` except that the string inserted is treated as a rectangle -- newline characters don't get inserted, instead the next line is inserted in the next line in the buffer at the same column as that inserted into the previous line. For more details see the function ``insert-rect'`.

At present there is a problem with changing the case of a rectangular block with ``Ctrl-x Ctrl-l'` or ``Ctrl-x Ctrl-u'`, they treat

it as a sequential block. This will be fixed soon.

1.40 jade.guide/Killing

Killing

=====

"Killing" is the general method for deleting a piece of text so that it can later be re-inserted into a buffer. Each time you kill some text it is stored in the window-system's clipboard (see see

Cutting And Pasting

) where it can be accessed by Jade or other programs.

The text copied by successive kill commands are concatenated together, this makes it easy to incrementally save text a piece at a time.

The main commands for killing are as follows, they are only described in brief since their full descriptions are in other parts of the manual.

'Ctrl-w'

Kill the current block. See
Using Blocks

.

'Meta-w'

Kill the current block without actually deleting it from the buffer.

'Ctrl-k'

Kills the current line. See
Editing Lines

.

'Meta-d'

Kill the word starting from the cursor. See
Editing Words

.

'Meta-Backspace'

Kills from the cursor to the beginning of the current word.

'Ctrl-Meta-k'

Kill the expression following the cursor. See
Editing Expressions

.

1.41 jade.guide/Searching and Replacing

Searching and Replacing

=====

It is very easy to search any of Jade's buffers for a specific string, the standard search command will search the current buffer for a specified regular expression.

Once you have found an occurrence of the string you are looking for it is then possible to replace it with something else.

Regular Expressions

The syntax of regular expressions

Incremental Search

How to search for regexps

Global Replace

Replacing all occurrences of a regexp

Query Replace

Interactively replacing regexps

1.42 jade.guide/Regular Expressions

Regular Expressions

Jade uses the `regexp(3)` package by Henry Spencer, with some modifications that I have added. It comes with this heading:

Copyright (c) 1986 by University of Toronto.
Written by Henry Spencer. Not derived from licensed software.

Permission is granted to anyone to use this software for any purpose on any computer system, and to redistribute it freely, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from defects in it.
2. The origin of this software must not be misrepresented, either by explicit claim or by omission.
3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software.

The syntax of a regular expression (or regexp) is as follows (this is quoted from the `regexp(3)` manual page):

A regular expression is zero or more "branches", separated by `|`. It matches anything that matches one of the branches.

A branch is zero or more "pieces", concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an "atom" possibly followed by `*`, `+`, or `?`. An atom followed by `*` matches a sequence of 0 or more matches of the atom. An atom followed by `+` matches a sequence of 1 or more matches of the atom. An atom followed by `?` matches a match of the atom, or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a "range" (see below), `.` (matching any single character), `^` (matching the null string at the beginning of the input string), `\$` (matching the null string at the end of the input string), a `\<` followed by a single character (matching that character), or a single character with no other significance (matching that character).

A "range" is a sequence of characters enclosed in `[]`. It normally matches any single character from the sequence. If the sequence begins with `^`, it matches any single character *not* from the rest of the sequence. If two characters in the sequence are separated by `-`, this is shorthand for the full list of ASCII characters between them (e.g. `[0-9]` matches any decimal digit). To include a literal `]` in the sequence, make it the first character (following a possible `^`). To include a literal `-`, make it the first or last character.

Some example legal regular expressions could be:

```
`ab*a+b'
```

Matches an `a` followed by zero or more `b` characters, followed by one or more `a` characters, followed by a `b`. For example, `aaab`, `abbbab`, etc...

```
`(one|two)_three'
```

Matches `one_three` or `two_three`.

```
`^cmd_[0-9]+'
```

Matches `cmd_` followed by one or more digits, it must start at the beginning of the line.

As well as being matched against, regexps also provide a means of "remembering" portions of the string that they match. The first nine parenthesised expressions that are matched and the whole match are recorded so that they can be used later.

The main use for this is in the command to replace a previously found regexp with the Lisp functions `regexp-expand`, `regexp-expand-line` and `replace-regexp`. The string which is given as the template (i.e. the string that replaces the matched string) is expanded inserting these recorded strings where asked to.

Each occurrence of `\

`&`

`0`

Replaces the whole substring matched by the regular expression.

`1` to `9`

The numbered parenthesised expression.

`\`

The character `\`.

For example, if a regexp of ``:([0-9]+):'` matches a line `'foo:123:bar'`, the expansion template `'x_\1'` would produce `'x_123'`.

1.43 jade.guide/Incremental Search

Incremental Search

Jade's main command for searching buffers is an Emacs-style incremental search (or "isearch"). This is a subsystem of the editor which lets you interactively search for regular expressions in a buffer.

`Ctrl-s`

Start an incremental search, initially searching forwards through the buffer.

`Ctrl-r`

Similar to `Ctrl-s` except that searching is initially in the backwards direction.

When you are in an isearch the general idea is to type in a regular expression and see what it matches in the buffer. As more characters are added to the string being searched for the cursor indicates strings which match. To backtrack your steps (i.e. the characters you have typed) the backspace key is used.

The special commands which are available when isearching are,

`Ctrl-s`

Search forwards for another occurrence of the search regexp. This can also be used to wrap around to the start of the buffer if no more matches exist between the cursor and the end of the buffer.

`Ctrl-r`

Search backwards for the regexp.

`Ctrl-g`

Cancels the isearch. If the search is currently failing (the string you've typed doesn't match anything) characters are deleted from the regexp until either a match is found or the original cursor position is reached. If the search is not failing the cursor is returned to its original position and the isearch is exited.

``Ctrl-w``

Copies the word under the cursor to the regexp being searched for.

``Ctrl-y``

The rest of the current line is appended to the regexp being searched for.

``Ctrl-q``

The next character typed is appended to the regexp no matter what it is, this can be used to enter control characters. Note that currently you can't search for newline characters.

``RET``

``ESC``

Accept the cursor's current position, the `isearch` is exited leaving the cursor as it is.

``Backspace``

Moves back up the stack which represents the current `isearch`, i.e. deletes characters from the search regexp or moves the cursor through the positions it had to reach its current position.

Any other keys are appended to the regular expression being searched for.

1.44 jade.guide/Global Replace

Global Replace

``Meta-x replace-all RET REGEXP RET TEMPLATE``

For all occurrences of the regular expression `REGEXP` replace it with the string obtained by expanding `TEMPLATE`. For details of how the `TEMPLATE` works see

Regular Expressions

.

1.45 jade.guide/Query Replace

Query Replace

The ``query-replace`` function provides an interactive method of replacing strings in a buffer which match a specific regular expression.

For each occurrence found you, the user, have a number of options; for example, you could replace this occurrence with a prespecified template.

``Meta-% REGEXP RET TEMPLATE RET'`

Invoke a query-replace, for all occurrences of the regular expression, REGEXP you will be prompted for what you want to do with it. Usually this will be to replace it with the expansion provided by the template (see see Regular Expressions) TEMPLATE.

Special commands which come into effect each time the query-replace finds a match are,

``SPC'`

``y'`

Replace this occurrence with the expansion of TEMPLATE and search for the next match.

``Backspace'`

``n'`

Ignore this match and search for the next.

``,``

Replace this occurrence, then wait for another command.

``RET'`

``ESC'`

``q'`

Exit this query-replace.

``.``

Replace this occurrence then exit.

``!'`

Replace the current match and all the rest between here and the end of the buffer.

`^^`

Retrace your steps through each match which has been found.

``Ctrl-r'`

Enter a recursive-edit, this is allows you to edit this match by hand. When you exit the recursive-edit (with the ``Ctrl-Meta-c'` command) the next match is searched for.

``Ctrl-w'`

Delete the current match, then enter a recursive-edit, as in the ``Ctrl-r'` command.

1.46 jade.guide/Editing Modes

Editing Modes

=====

Modes are used to tailor the editor to the **type** of the file being edited in a buffer. They are normally a file of Lisp which installs the buffer-local key bindings and variables which are needed for that type of file.

For example, C-mode is a mode used to edit C source code, its main function is to try to indent each line to its correct position automatically.

The name of the mode active in the current buffer is displayed in the status line, inside the square brackets.

At present there are only a small number of modes available. It is fairly straightforward to write a mode for other classes of file though. See

Writing Modes

.

Most of the modes for editing programming languages use the command `'Meta-;'` to insert a comment place-holder, the cursor is moved to where you should type the body of the comment.

Invoking a Mode

How editing modes are invoked on a buffer

Generic mode

The foundations which all modes build from

-- Modes for editing programming languages --

C mode

Mode for C source code

Lisp mode

Mode for Lisp

Asm mode

For generic assembler source

-- Modes for natural language --

Text mode

For normal language-based text

Indented-Text mode

Variant of Text-mode

Texinfo mode

Mode for editing Texinfo source

1.47 jade.guide/Invoking a Mode

Invoking a Mode

When a new file is loaded the function `'init-mode'` tries to find the mode that it should be edited with. If it is successful the mode will be automatically invoked.

It is also possible to install a mode manually, simply invoke the command which is the name of the mode. For example to install the `'C mode'` in a buffer type `'Meta-x c-mode'`.

1.48 jade.guide/Generic mode

Generic mode

This is not a mode as such since there is no Lisp code associated with it. When no mode is being used to edit the buffer, it is said to use the "Generic" mode.

This is the base from which all other modes build, it consists of all the standard key bindings. Words are defined as one or more alphanumeric characters, paragraphs are separated by a single blank line.

1.49 jade.guide/C mode

C mode

`'c-mode'` is used for editing C source code files. Any files which end in `'c'` or `'h'` are automatically edited in this mode.

It's one and only function is to try and indent lines to their correct depth, it doesn't always get it right but it works fairly well. The keys that it rebinds to achieve this are,

It also defines the syntax of an expression in the C language for use with the expression commands, see

Editing Expressions

.

`'TAB'`

Indents the current line to what the editor thinks is the correct position.

`'{'`
`'}'`

``:'`

These keys are handled specially since the indentation of the line that they are inserted on may have to be adjusted.

``Ctrl-Meta-\'`

Indents all lines which are marked by the current block.

- Command: `c-mode`

Editing mode for C source code. Automatically used for files ending in ``.c'`` or ``.h'``.

- Hook: `c-mode-hook`

This hook is called by `'c-mode'` each time the mode is invoked.

- Variable: `c-mode-tab`

Size of tab stops used by `'c-mode'`.

- Variable: `c-mode-auto-indent`

When non-nil `'RET'` will indent the line after splitting it.

1.50 jade.guide/Lisp mode

Lisp mode

`'lisp-mode'` is used to edit files of Lisp intended to be read by the editor. Its main function is to manage the indentation of Lisp expressions for you. Each form is regarded as an expression by the commands which operate on expressions, see

Editing Expressions

.

There is also support for using a buffer as a simple shell-interface to the editor's Lisp subsystem.

The method used for indenting lines of Lisp is fairly straightforward, the first symbol in the expression containing this line is found. This symbol's `'lisp-indent'` property is then used to decide which indentation method to apply to this line. It can be one of the following,

`'nil'`

The standard method (also used if the symbol doesn't have a `'lisp-indent'` property).

If the first argument to the function is on the same line as the name of the function then subsequent lines are placed under the first argument. Otherwise, the following lines are indented to the same depth as the function name.

For example,

```
(setq foo 20
```

```
bar 1000)
```

```
(setq
  foo 20
  bar 1000)
```

'defun'

This method is used for all functions (or special-forms, macros) whose name begins with 'def' and any lambda-expressions.

All arguments to the function are indented 'lisp-body-indent' columns from the start of the expression.

For example,

```
(defun foo (bar)
  "A test"
  (let
    ((foo bar))
    ...
```

A number, N

The first N arguments to the function are indented twice the value of 'lisp-body-indent', the remaining arguments are indented by 'lisp-body-indent'.

For example the special-form 'if' has a 'lisp-indent' property of 2,

```
(if condition
  t-expression
  nil-expressions...)
```

Special commands for Lisp mode are,

'Ctrl-j'

Evaluates the expression preceding the cursor, prints the value on the next line. This is designed to be used like a shell, you type a Lisp expression, press 'Ctrl-j' and Jade prints the value for you.

'TAB'

Indents the current line.

'Ctrl-Meta-\'

Indents all lines which are marked by the current block.

'Ctrl-Meta-x'

Evaluates the expression before the cursor, prints it's value in the status line.

- Command: lisp-mode

Editing mode for Jade's Lisp. Automatically invoked for files ending in '.jl'.

- Hook: lisp-mode-hook

This hook is evaluated each time 'lisp-mode' is invoked.

- Variable: `lisp-body-indent`
The number of characters which the body of a form is indented by, the default value is 2.

1.51 jade.guide/Asm mode

Asm mode

A basic mode for editing assembler source files with, provides automatic indentation of labels and instructions.

The special commands are,

``RET'`

Breaks the line as normal, if `'asm-indent'` is non-nil a tab character is inserted as well.

`':'`

Deletes all indentation from the start of the current line, then inserts the string `':\t'` to move to the next tab stop. This is used to enter labels.

``.``

If the line is not empty, all indentation is deleted from the start of the line. A dot (``.``) is then inserted.

- Command: `asm-mode`

Major mode for generic assembler source files.

- Hook: `asm-mode-hook`

The hook which is called when `'asm-mode'` is entered.

- Variable: `asm-indent`

When this variable is non-nil the RET key inserts the string `'\n\t'` instead of just `'\n'`. This indents the cursor to the first tab stop of the new line.

- Variable: `asm-comment`

This variable defines the string which denotes the start of a comment in the assembler that you are using. By default this is `';'` .

1.52 jade.guide/Text mode

Text mode

This is the most basic mode for editing English-style text in. The main difference over 'generic-mode' and is that words are allowed to contain underscores and there are some extra commands,

'Meta-s'

Centres the current line. The position of the 'fill-column' is used to calculate the centre of the line. For more information on the 'fill-column' variable see
Fill mode
.

'Meta-S'

Centres the current paragraph.

- Command: text-mode

Major mode for editing English text.

- Hook: text-mode-hook

Evaluated when 'text-mode' is invoked. Variants of 'text-mode' also use this hook.

1.53 jade.guide/Indented-Text mode

Indented-Text mode

This is a variant of 'text-mode', see

Text mode

. It's only

difference is in the way the TAB key is handled -- tab stops are calculated from the previous non-empty line. Each transition from a sequence of one or more spaces to a non-space character is used as a tab stop. If there are none of these to the right of the cursor normal the standard tabbing command is used.

- Command: indented-text-mode

Variant of 'text-mode'.

- Hook: indented-text-mode-hook

Evaluated when 'indented-text-mode' is invoked. The hook 'text-mode-hook' is also evaluated (before this one).

1.54 jade.guide/Texinfo mode

Texinfo mode

'texinfo-mode' is used to edit Texinfo source files, it is

automatically selected for files ending in `.texi` or `.texinfo`. It provides a few basic key bindings to take some of the tedium out of editing these files.

Paragraphs are separated by the regexp `^@node`, i.e. each node is a separate paragraph.

The special commands are,

``TAB'`

Inserts as many spaces as are needed to move the cursor to the next tab position. The reason tab characters aren't used is that TeX doesn't agree with them.

``Ctrl-c Ctrl-c c'`

Insert the string `@code{}`, positioning the cursor between the braces.

``Ctrl-c Ctrl-c d'`

Insert the string `@dfn{}`, positioning the cursor between the braces.

``Ctrl-c Ctrl-c e'`

Inserts the string `@end`.

``Ctrl-c Ctrl-c f'`

Inserts the string `@file{}`, the cursor is put between the braces.

``Ctrl-c Ctrl-c i'`

Inserts the string `@item`.

``Ctrl-c Ctrl-c l'`

Inserts the string `@lisp\n`.

``Ctrl-c Ctrl-c m'`

Inserts the string `@menu\n`.

``Ctrl-c Ctrl-c Ctrl-m'`

Prompts for the name of a node and makes a menu-item for it.

``Ctrl-c Ctrl-c n'`

Prompts for each part of a node definition (name, next, prev, up) and inserts the `@node ...` string needed.

``Ctrl-c Ctrl-c s'`

Inserts the string `@samp{}` and puts the cursor between the braces.

``Ctrl-c Ctrl-c v'`

Inserts the string `@var{}`, the cursor is put between the braces.

``Ctrl-c Ctrl-c {'`

Inserts a pair of braces with the cursor between them.

``Ctrl-c Ctrl-c }'`

``Ctrl-c Ctrl-c]'`

Moves the cursor to the character after the next closing brace.

- Command: `texinfo-mode`
Major mode for editing Texinfo source files.
- Hook: `texinfo-mode-hook`
Evaluated when `'texinfo-mode'` is invoked. The hook `'text-mode-hook'` is evaluated first.

1.55 jade.guide/Minor Modes

Minor Modes

=====

The editing modes described in the previous section were "Major modes", each mode was designed for a particular class of file. Minor modes work on top of the major modes, each minor mode provides a single extra feature for editing the buffer they are used in. For example `'overwrite-mode'` is a minor mode which makes any keys you type overwrite the character beneath the cursor, instead of inserting themselves before the cursor.

The names of the minor modes currently active in the current buffer are displayed in the status line, to the right of the name of the major mode.

Overwrite mode

Typed characters overwrite the character beneath them.

Fill mode

Automatically break long lines as they are typed.

Auto-Save mode

How to disable auto-saving of a buffer.

Latin-1 mode

Displaying European characters.

1.56 jade.guide/Overwrite mode

Overwrite mode

When enabled, characters typed replace the existing character under the cursor instead of just moving it to the right.

The command to toggle this mode on and off is `'Meta-x overwrite-mode'`.

- Command: `overwrite-mode`
Toggles overwriting character insertion in the current buffer.

1.57 jade.guide/Fill mode

Fill mode

Filling splits lines so that they aren't longer than a certain number of characters. The `'fill-mode'` checks if you have passed this threshold when you type the SPC key. Any words passed the threshold get moved to the next line.

`'Ctrl-x f'`

Sets the `'fill-column'` variable (see below) to the cursor's current column position.

- Command: `fill-mode`
Toggles the auto-filling minor mode.
- Variable: `fill-column`
The maximum number of characters allowed in a single line. This is used by the filling and centring functions.

1.58 jade.guide/Auto-Save mode

Auto-Save mode

This is not really a minor mode but it obeys the same calling conventions (i.e. calling its function toggles its action).

- Command: `auto-save-mode`
Toggles whether or not the current buffer is regularly saved to a temporary file.

For more details about auto-saving see
Auto-Saving Files

.

1.59 jade.guide/Latin-1 mode

Latin-1 mode

This minor mode toggles the display of characters in the Latin-1 character set, by default these characters are displayed as octal escape sequences.

This only works properly if the font that you are using defines glyphs for these characters!

- Command: latin-1-mode
Toggles the display of characters in the Latin-1 character set.
This is a *global* setting.

For more information about what is displayed for each character see

Character Images

.

1.60 jade.guide/Using Buffers

Using Buffers

=====

As you have probably realised, buffers are probably the most important part of the editor. Each file that is being edited must be stored in a buffer. They are not restricted to editing files though, all buffers are regarded as simply being a list of lines which can be displayed in a window and modified as needed.

This means that they are very flexible, for example, the Lisp debugger uses a buffer for its user interface, the Info reader uses two buffers - one to display the current node, the other to store the file's tag table (never displayed, just used to look up the position of nodes).

Each buffer has a name, generally buffers which contain proper files use the base part of the filename, while buffers which don't correspond to files use a word which starts and ends with asterisks (i.e. '*jade*').

Each window can display one buffer at any one time. There is no restriction on the number of windows which may display the same buffer at once.

Displaying Buffers

How to make a window display a buffer

Deleting Buffers

Killing unwanted buffers

Other Buffer Commands
 General buffer manipulation

The Buffer Menu
 Interactive buffer manipulation

1.61 jade.guide/Displaying Buffers

Displaying Buffers

There are two main commands for switching to a different buffer,

``Ctrl-x b'`

Prompt for the name of a buffer and display it in the current window.

``Ctrl-x 4 b'`

In a different window (opens a new window if there is currently only one) prompt for the name of a buffer and display it in that window.

Both commands are very similar, the ``Ctrl-x 4 b'` variant simply invokes a command to switch to a different window before calling the ``Ctrl-x b'` command.

When typing the name of the new buffer you can use the prompt's completion mechanism to expand abbreviations (see see

The Buffer Prompt

). If you just press RET with an empty prompt the default choice will be used. This will be the the buffer that was being shown in this window before the current buffer was selected (its name is displayed in the prompt's title).

The ``Ctrl-x Ctrl-f'` command and its variants also switch buffers since they look for an existing copy of the file in a buffer before loading it from disk, see

Commands To Load Files

.

1.62 jade.guide/Deleting Buffers

Deleting Buffers

There is no real need to delete buffers, those that haven't been

used for a while just hang around at the end of the list. If you're short on memory though it can help to kill some of the unused buffers which you have accumulated.

The command to kill a buffer is,

``Ctrl-x k'`

Prompts for the name of a buffer (with completion) then deletes that buffer (if the buffer contains unsaved modifications you are asked if you really want to lose them). It is removed from all window's buffer-lists and any window which is displaying it is switched to another buffer (the next in its list).

Any marks which point to the buffer are made "non-resident" (that is, they point to the name of the file in the buffer) and the buffer is discarded.

1.63 jade.guide/Other Buffer Commands

Other Buffer Commands

``Meta-x rotate-buffers-forward'`

Rotates the current window's list of buffers.

``Meta-x revert-buffer'`

Restores the contents of the current buffer to the contents of the file that it was loaded from, if an auto-save file exists you are asked if you want to revert to that instead.

``Ctrl-x s'`

Ask whether to save any modified buffers that exist.

``Meta-x clear-buffer'`

Deletes the contents of the current buffer. Beware, you **won't** be warned if you're about to lose any unsaved modifications!

1.64 jade.guide/The Buffer Menu

The Buffer Menu

The buffer menu presents you with a list of all the buffers accessible from the current window in most-recently-used order. You are then able to manipulate the buffer list using several simple commands.

``Ctrl-x Ctrl-b'`

Enters the buffer menu; the buffer ``*Buffer Menu*'` is selected and a list of available buffers is printed in it.

The following example shows how the buffer list is printed.

MR	Name	Mode	File
--	----	----	----
-	*Buffer Menu*	Buffer Menu	
+	user.texi	Texinfo	man/user.texi
	jade	Lisp	

The column headed 'M' shows whether the buffer has been modified since it was last saved and the column 'R' shows whether or not the buffer is read-only. The other columns should be self-explanatory.

When the '*Buffer Menu*' buffer is selected the following commands are available. When a single buffer is to be manipulated by a command, the buffer described by the line which the cursor is on is chosen.

'd'

Mark the buffer for deletion and move to the next buffer. A 'D' is displayed in the first column of a line if that buffer is marked for deletion.

's'

'Ctrl-s'

Mark the buffer to be saved then move to the next buffer in the list. A 'S' in the second column of a line denotes a buffer which has been marked to be saved.

'x'

Execute previously-marked saves and deletions.

'u'

Unmark the current line (i.e. clear any 'D' or 'S' markers) then move to the next entry in the buffer list.

'~'

Toggle the modified flag of the current line's buffer, then move down.

'%'

'_'

Toggle the read-only status of the current line's buffer, then move to the next entry.

'l'

'RET'

Select the current line's buffer in this window.

'o'

Select the current line's buffer in the other window.

'Ctrl-f'

'TAB'

Move to the next line in the buffer list.

'Ctrl-b'

Move to the previous line in the buffer list.

``Ctrl-l'`

Redraw the buffer list, incorporating any changes made to the available buffers.

``q'`

Quit the buffer menu.

1.65 jade.guide/Using Windows

Using Windows

=====

Windows have two main functions: to display the contents of buffers (but only one buffer at a time) and to collect input from you, the user.

The editor *must* have at least one window open at all times, when you close the last window Jade will exit, there is no limit to the number of windows which you may have open at once.

Each window is split into two parts, they are

"The Main Display Area"

This is the largest part of the window, it is where the buffer that this window is displaying is drawn.

"The Status Line"

A single line of text associated with the window, under X11 this is the area of the beneath the horizontal line at the bottom of the window, on the Amiga it is the title of the window. The status line is normally used to display information about this window and what it is displaying, it has this format,

```
BUFFER-NAME (MODE-NAMES) (COL,ROW) N line(s) [FLAGS]
```

Where the individual parts mean,

BUFFER-NAME

The name of the buffer being edited, it can have either a '+' or a '-' appended to it, a plus means the buffer has been modified since it was saved, a minus means that the buffer is read-only.

MODE-NAMES

This tells you which editing modes are being used by this buffer, the first word is the name of the major mode, any subsequent words correspond to the names of the minor modes for this buffer. If this section is surrounded by square brackets '[...]' instead of parentheses it means that you are currently in a recursive edit, for example, inside the Lisp debugger.

COL

The column that the cursor is at.

ROW

The row number of the cursor.

N

The number of lines in this buffer

FLAGS

General one-character flags related to the status of the window and its buffer.

Each window maintains a list of all buffers which are available for displaying, this is kept in order, from the most recently used to the least. This list (called 'buffer-list') is used by some of the buffer manipulation commands when they are working out which buffer should be displayed.

Creating Windows

Opening a new window

Killing Windows

How to close windows

Other Window Commands

General window manipulation

1.66 jade.guide/Creating Windows

Creating Windows

'Ctrl-x 2'

Opens a new window, it will have the most of the attributes that the current window does, things like: size, buffer, font, etc... If you are using X11 you will probably have to use your mouse to select its position, depending on the window manager you use, on the Amiga it will be created at the same position as the current window.

'Ctrl-x 4 Ctrl-f'

'Ctrl-x 4 f'

In a different window, one will be created if only one window is open, find a file, for more details see
Commands To Load Files

.

'Ctrl-x 4 a'

In a different window add an entry to a change-log file. See

Keeping ChangeLogs

.

``Ctrl-x 4 b'`

In a different window, choose a buffer to display, similar to the ``Ctrl-x b'` command. See
Displaying Buffers

.

``Ctrl-x 4 h'`

Enter the help system in a different window. See
The Help System

.

``Ctrl-x 4 i'`

Enter the Info browser in a different window. See
Info Mode

.

``Ctrl-x 4 `'`

Display the next error (or whatever) in the ``*compilation*'` buffer in a different window. See
Finding Errors

.

Note that for each ``Ctrl-x 4'` command there is a corresponding ``Ctrl-x 5'` command. Instead of using a different window to the current one, a new window is opened for each ``Ctrl-x 5'` command typed.

1.67 jade.guide/Killing Windows

Killing Windows

``Ctrl-x 0'`

Close the current window, if it is the last window that the editor has open it will exit (after asking you if you wish to lose any unsaved modifications to buffers).

``Ctrl-x 1'`

Close all windows except the current one.

1.68 jade.guide/Other Window Commands

Other Window Commands

``Ctrl-x o'`

Activate the next window of the editor's. Under X11 this involves warping the mouse-pointer to the top left corner of the newly activated window.

``Meta-x set-font'`

Choose a font to use in the current window. This command prompts for the name of the font then installs it in the window. Font names are the same as for the shell argument `'-font'` (see

Startup Options
).

1.69 jade.guide/Using the Prompt

Using the Prompt

=====

There are two different styles of prompt that the editor uses when it wants you to enter a string.

The Simple Prompt

The prompt at the bottom of the window

The Buffer Prompt

Prompt with its own buffer and completion

1.70 jade.guide/The Simple Prompt

The Simple Prompt

The simplest prompt uses the the bottom-most line in the window, it prints the prompt's title on the left hand side, you should type your response and then press the RET key. This prompt is very primitive, the only special commands that it has are,

``Backspace'`

Delete the previous character.

``Up'`

``Down'`

Replace the contents of the prompt with the last string entered. When you type `'Up'` or `'Down'` again the original contents are restored.

``ESC'`

Cancel the prompt.

All other keys are simply printed in the prompt -- whatever they are.

1.71 jade.guide/The Buffer Prompt

The Buffer Prompt

This type of prompt is more sophisticated. It creates a new buffer for you to type your response into (called `*prompt*`), the title of the prompt is displayed in the buffer's first line.

Normally you type the answer to the prompt into the buffer and then press the RET key. All normal editor commands are available while you are using the prompt, you can switch buffers, load new files, whatever you like.

Another advantage of this type of prompt is that it supports "completion", this allows you to type the beginning of your response then press the TAB key. What you have typed will be matched against the list of responses that the editor has (i.e. when being prompted for the name of a file it will be matched against all available files), if a unique match is found your response will be completed to that match.

If several potential completions are found, these will be displayed after the line `::Completions::` in the buffer and your response will only be completed as far as the potential completions are similar. For example, if you enter `fo` then press TAB and files called `foo` and `foobar` exist, the contents of the prompt will become `foo`.

Completion is provided for many different things, some are: files, buffers, symbols, functions, variables, Info nodes, etc...

The special commands for this type of prompt are,

`'TAB'`

`'RMB-CLICK1'`

Complete the contents of the prompt. If more than one potential completion exists they are printed in the buffer.

`'RET'`

`'LMB-CLICK2'`

Enter the result of this prompt. If you invoke this command while the cursor is on a printed potential completion (those under the `::Completions::` line) the whole line will be entered. Otherwise, just the text to the left of the cursor is entered.

`'Meta-?'`

Print all possible completions of the current prompt but do not try to actually change the contents of the prompt.

`'Ctrl-g'`

Cancel the prompt.

1.72 jade.guide/Using Marks

Using Marks

=====

Marks are used to record a position in a file, as the file's buffer is modified so does the position that the mark points to -- a mark will keep pointing at the same character no matter what happens (unless the character is deleted!).

The other good thing about marks is that they point to files **not** buffers. This means that you can set a mark in a buffer, delete the buffer and then move to the position of the mark, the file will be reloaded and the cursor will point at the original character.

Normally there are three user-accessible marks (1) and one special 'auto-mark' which is used, amongst other things, to record the "previous" position of the cursor, allowing you to retrace your last major step.

The commands available on marks are,

'F1'

'F2'

'F3'

Move to the mark #1, #2 or #3, depending on which function key is pressed (F1 means mark #1, etc...). If the file pointed to is not in memory it will be loaded into a new buffer.

'Shift-F1'

'Shift-F2'

'Shift-F3'

Set the position of mark #1, #2 or #3, depending on the function key.

'Ctrl-x Ctrl-x'

Swap the positions of the cursor and the 'auto-mark'.

'Ctrl-@'

Set the position of the 'auto-mark'.

----- Footnotes -----

(1) There is no reason why you can't have more, the editor sets no limitation on the number of marks available. This is just how I have set the editor up.

1.73 jade.guide/Interrupting Jade

Interrupting Jade

=====

It is often useful to be able to tell Jade to quit whatever it is doing and wait for more commands; this is called "interrupting" Jade. When the editor receives an interrupt signal it will abort what it is

doing and rewind itself back to the inner-most recursive edit (see see

```
Recursive Editing
).
```

The interrupt signal differs with the operating system being used,

- * Under Unix the 'SIGINT' signal is used, this can be sent via the 'intr' character (get the editor into the foreground of the shell it was started from and type 'Ctrl-c' in the shell's terminal), or directly through the 'kill' shell command. For example, look at the following shell session extract,

```
/var/src/jade/man$ ps
PID TT STAT  TIME COMMAND
  60  1 SW   0:02 (xinit)
  87  1 S    0:08 fvwm
 127 p0 S    0:00 /bin/bash
 155 p0 S    0:04 jade
 156 p1 S    0:00 /bin/bash
 159 p1 R    0:00 ps
/var/src/jade/man$ kill -INT 155
```

First the 'ps' command is used to find the Jade process' pid (155), then the 'kill' command is used to send the 'INT' signal to this process.

- * The 'Ctrl-c' signal is also used on Amigas, either type this in the console window that Jade was launched from or use the 'break' (or possibly 'breaktask') command to send the signal.

1.74 jade.guide/Recursive Editing

Recursive Editing

=====

Recursive editing is the act of editing a file while the current command is still being evaluated. For example, when using the 'query-replace' command ('Meta-%') the 'Ctrl-r' command enters a recursive edit to let you edit the buffer, even though you are still doing a query-replace (which will be resumed when the recursive edit finishes).

As the name suggests a recursive edit calls the editor's main command loop recursively from within a command. Any number of recursive edits may be stacked up and then unwound back to the top-level of the editor.

When a recursive edit is in progress the name of the mode being used to edit the buffer is shown in *square brackets*, not parentheses as in the top-level instance.

The commands for manipulating recursive edits are as follows,

``Ctrl-j``

``Ctrl-Meta-c``

Exit the innermost recursive edit, this has no effect at the top-level.

``Meta-x top-level``

Return to the outermost edit -- the top-level. This is useful when you get "lost" inside a sequence of recursive edits.

``Meta-x recursive-edit``

Enter a new recursive edit; this command is usually best avoided to save confusion.

In general, recursive editing is rarely used except in unavoidable circumstances (i.e. in the Lisp debugger).

1.75 jade.guide/Character Images

Character Images

=====

In general any character can be mapped to any sequence of up to four character sized images (called glyphs) when it is drawn into a window. The TAB character is a notable exception; it expands to as many spaces as are needed to fill up to the next tab stop.

By default, the editor is set up to display the following,

0 to 31

A caret (``^``) followed by the ASCII value of the character exclusive-or'd with 0x40, i.e. ``^@`` to ``^_``.

32 to 126

Printed literally, this includes all "normal" characters and punctuation.

127

``^?``

128 to 255

Represented by the octal escape sequence (i.e. ``\200``) for that character's numeric value.

If you want to edit files containing characters in the 'Latin1' character set (numerically, from 160 to 255) you can put the following in your ``.jaderc`` file,

```
(latin-1-mode)
```

this will redefine the necessary characters.

If you want more details about this sort of thing see
Glyph Tables

.

1.76 jade.guide/Client Editing

Client Editing

=====

Normally you will only have one instance of Jade executing at a single time. Often though, another program will want you to edit a file, for example when you are composing a mail message. There is normally a way to specify which editor you want to use, for example the 'EDITOR' environment variable.

If you were to ask to edit the file in 'jade' an *additional* process executing Jade would be started, totally separate from the original. It is possible to use the original instance.

Firstly Jade must be set up to listen for clients wanting files edited, this is done with the 'server-open' command. You can either put this in your '.jaderc' file (with a line like '(server-open)') or call it manually with the command 'Meta-x server-open'.

Only one instance of Jade may be a server at once. If you know that there is no other Jade running but it still won't let you open a server, and you are running on Unix, look for a dead socket called '~/.Jade_rendezvous' and delete it if necessary.

Once the editor is listening for client messages the separate program 'jadeclient' may be used to load files into the server from an external source. The format of 'jadeclient' invocation is,

```
jadeclient [+LINE-NUMBER] FILE-NAME ...
```

When invoked, it will ask the server to edit each FILE-NAME (initially positioned at line LINE-NUMBER) in turn, exiting only after each file has finished being edited.

If when the 'jadeclient' program is invoked there is no server open (i.e. either Jade is not running or you haven't used the 'server-open' function) a message 'Jade not running, waiting...' will be printed and 'jadeclient' will sit waiting for you to open a Jade server.

So, simply get the program you want to use Jade to use the 'jadeclient' program as its editor. For example, I use 'mh' to handle my electronic mail; in my '~/.mh_profile' file I have the line,

```
Editor: jadeclient
```

to tell it that I want to edit my mail in Jade.

The one special command for client/server editing is,

```
'Ctrl-x #'
```

If the file being edited in the current buffer is a client file,

tell the client program which loaded it that it has finished being edited. The actual buffer is **not** deleted.

It is also possible to finish editing a client file by simple deleting its buffer in the normal way (`'Ctrl-x k'`),

Deleting Buffers

.

1.77 jade.guide/Compiling Programs

Compiling Programs

=====

Jade has a number of features to help you develop programs, foremost is the ability to run a compilation inside one of the editor's buffers. Unfortunately, this is only possible when using the Unix operating system at the present.

Once the compilation has finished you can then step through each error produced.

Running a Compilation

 Launching a compilation process

Finding Errors

 Stepping through compile errors

Debugging Programs

 Using GDB in an editor buffer

Using Grep

 Searching files for a regexp

Keeping ChangeLogs

 Simple recording of file revisions

1.78 jade.guide/Running a Compilation

Running a Compilation

The command to run a shell command in a buffer is,

`'Meta-x compile'`

Prompts you for the command to execute, with a default of the last command you ran (starts as `'make'`). A shell process is created

which runs asynchronously to the editor in the same directory as the current buffer's file was loaded from. The buffer `'*compilation*'` is selected and this is where all output from the program is printed.

When the process finishes running a message is printed in the `'*compilation*'` buffer telling you its exit-code.

Only one process may be run with the `'compile'` function at once.

This command is not available on the Amiga version yet.

1.79 jade.guide/Finding Errors

Finding Errors

When you have compiled something with the `'Meta-x compile'` command it is possible to step through each of the errors that it produces. To do this use the command,

`'Ctrl-x '`

Displays the next error in the `'*compilation*'` buffer. The file that is in is loaded (if necessary) and the line with the error is found.

If you edit a file which has errors in it, then try to find the next error (which is in the same file) everything will still work. The positions of errors are updated as the buffers are modified.

The only exception to this is when you invoke the `'next-error'` function while the `'*compilation*'` buffer is still being written to. If more errors are produced in a file which has been modified since the compilation started it is likely that the positions will get out of sync.

By default, the `'next-error'` function understands the type of error output that `'gcc'` produces. This is of the form,

FILE:LINE-NUMBER:DESCRIPTION

It is possible to use other formats though, the variables which control this are,

- Variable: `compile-error-regexp`
Regular expression to match a line containing an error. For `'gcc'` this is `'^(.*):[0-9]+:(.+)'`.
- Variable: `compile-file-expand`
Expansion template to produce the name of the file with the error, using `'compile-error-regexp'` and the line containing the error. By default this is `'\1'`.
- Variable: `compile-line-expand`

Similar to `'compile-file-expand'` except that it expands to a string defining the number of the line with the error. By default, `'\2'`.

- Variable: `compile-error-expand`
Similar to `'compile-file-expand'`, but produces the description of the error. By default, `'\3'`.

1.80 jade.guide/Debugging Programs

Debugging Programs

Jade allows you to run the GDB debugger in a buffer. Some of the advantages of this over the usual terminal based interaction are,

- * The current position of the target program (its "frame") is highlighted; the source file is displayed in a separate window with the current frame marked (in the same way that a block is marked).
- * You are able to set and delete breakpoints simply by putting the cursor on the line you wish the target to stop at and typing an editor command.

To start a gdb subprocess use the `'Meta-x gdb'` command, you will be asked to enter the name of the program to debug then gdb will be started in a new buffer (called `'*gdb*' or similar`). You are then able to type commands into the buffer, they will be sent to gdb each time you type the RET key.

The commands for controlling the gdb subprocess are as follows (the `'Ctrl-c'` prefixed commands are only available within the `'*gdb*' buffer whereas the 'Ctrl-x Ctrl-a' variations are accessible globally so that they can be invoked from within the target's source files),`

`'Ctrl-c Ctrl-n'`

`'Ctrl-x Ctrl-a Ctrl-n'`

Continue execution to the next source line, this is the gdb command `'next'`.

`'Ctrl-c Ctrl-s'`

`'Ctrl-x Ctrl-a Ctrl-s'`

Continue execution until a different source line is reached, this is the gdb command `'step'`.

`'Ctrl-c Ctrl-f'`

`'Ctrl-x Ctrl-a Ctrl-f'`

Continue running until the current stack frame exits, the `'finish'` command.

`'Ctrl-c Ctrl-r'`

`'Ctrl-x Ctrl-a Ctrl-r'`

Resume execution until a breakpoint is reached or the target exits.

```
`Ctrl-c Ctrl-<`
`Ctrl-x Ctrl-a Ctrl-<`
    Display the stack frame above the current one.

`Ctrl-c Ctrl->`
`Ctrl-x Ctrl-a Ctrl->`
    Display the stack frame under the current one.

`Ctrl-c Ctrl-b`
`Ctrl-x Ctrl-a Ctrl-b`
    Set a breakpoint at the current source line, if the `*gdb*` buffer
    is active the line selected is where the program last stopped.

`Ctrl-c Ctrl-t`
`Ctrl-x Ctrl-a Ctrl-t`
    Set a temporary breakpoint at the current source line.

`Ctrl-c Ctrl-d`
`Ctrl-x Ctrl-a Ctrl-d`
    Remove all breakpoints which are set at the current source line.

`Ctrl-c Ctrl-l`
`Ctrl-x Ctrl-a Ctrl-l`
    Redisplay the current frame, centring it in its window.
```

For a summary of these commands type `Ctrl-h m` in the `*gdb*` buffer.

Since the gdb process runs on top of the Shell mode the bindings from that mode are also available.

There is no limit to the number of gdb processes you may run at once, each will get its own buffer. When a gdb command is invoked in a buffer which doesn't have a gdb subprocess (i.e. a source file's buffer) the command will be sent to the gdb process which either was last sent a command, or last made the editor display a new frame. Hopefully this will work fairly intuitively.

1.81 jade.guide/Using Grep

Using Grep

It is often very useful to grep through a set of files looking for a regular expression, this is what the `grep` command does. With Jade it is possible to run an external `grep` program in the `*compilation*` buffer. This then enables you to step through each grep hit using the `Ctrl-x ` command,

Finding Errors

.

The commands to use grep are,

```
`Meta-x grep`
```

Prompt for a string of arguments to give 'grep', you do not need to provide the name of the program, or the '-n' switch, this is done automatically. The shell will do any filename-globbing on the arguments so it is advisable to surround the regular expression with single quotes.

Note that the regular expression syntax will be different to that which Jade uses. Also this command won't work on an Amiga.

'Meta-x grep-buffer'

This command provides a method for scanning the current buffer for all lines matching a regular expression (which you are prompted for). It is written entirely in Lisp -- this means that the normal regular expression syntax is needed and it will work on an Amiga.

1.82 jade.guide/Keeping ChangeLogs

Keeping ChangeLogs

A ChangeLog is a file (usually called 'ChangeLog') which keeps a log of all changes you have made to the files in its directory. For example, the 'src/ChangeLog' file for Jade keeps a list of changes made to the editor's source code.

There is no magic involved, you simply use a command to add a new entry to a directory's log after modifying a file in that directory. You then have to enter a summary of the changes that you made.

The command to do this is,

'Meta-a'

Prompts for the name of a directory then lets you type a description of the changes you have made.

If you enter more than one change in the same day (and from the same host) the same heading will be used. The heading consists of the time and date, your name, your login and the name of the host you're on. (1)

----- Footnotes -----

(1) On the Amiga there is no way to get these details. So, Jade looks for some environment variables, 'USERNAME' for the login name, 'HOSTNAME' for the name of the host and 'REALNAME' for your actual name.

1.83 jade.guide/Info Mode

Info Mode

Despite the name of this section there is actually no such thing as the 'info-mode'. The Lisp file 'info.jl' is what this section documents -- it is a set of Lisp functions which make a buffer (the '*Info*' buffer) into a simple browser for Info files(1).

To invoke it type 'Ctrl-h i', the '*Info*' buffer will be selected showing the '(dir)' node (the root of the Info documentation tree).

When in the '*Info*' buffer the following key bindings are available.

'SPC'

Displays the next page of the current node.

'Backspace'

Displays the previous page.

'1'

Move to the specified menu-item ('1' means the first, etc) in the menu in this node. The keys '1' to '9' work in this way.

'b'

Move to the beginning of the current node.

'd'

Display the directory node ('(dir)') of the Info documentation tree.

'f'

Follow a reference, the one under the cursor if one exists.

'g'

Prompt for the name of a node and try to display it.

'h'

Display the Info tutorial node ('(info)Help').

'l'

Go back to the last node that was displayed before this one.

'm'

Prompts for a menu-item (the one on the same line as the cursor is the default) and display the node it points to.

'n'

Display the next node.

'p'

Display the previous node.

'u'

Display the node "above" this one.

'q'

Quit the Info browser.

'?'

Display a piece of text describing all commands available in Info mode.

`'RET'`

Go to the link (menu item or xref) described on the current line.

`'LMB-Click2'`

Go to the link you double clicked on.

`'TAB'`

Put the cursor on the next link in this node.

`'Meta-TAB'`

Put the cursor on the previous link.

This mode has a number of disadvantages over the other Info browsers available (i.e. the stand-alone 'info' program, or Emacs' Info viewer):

- * It depends wholly on being able to find a tag table in the Info file, if it can't it will simply load the whole file into the buffer.
- * There is no support for the '*' node name.
- * Seems not to work 100% with files formatted by Emacs, 'makeinfo' formatted files work properly though.
- * No editing of nodes.

Of course, its main advantage is that it runs in Jade!

----- Footnotes -----

(1) 'Info' is the GNU way of creating hypertext documents, for more information see Info.

1.84 jade.guide/Shell

Shell

=====

When running on a Unix-style operating system Jade allows you to run a shell subprocess in a buffer (usually the '*shell*' buffer). Each line you type in the buffer is sent to the shell and the output from the shell is displayed in the buffer.

`'Meta-x shell'`

Start a new shell subprocess running in a buffer called '*shell*'.

If a buffer '*shell*' already exists a new buffer with a unique name will be opened (i.e. '*shell*<2>').

The working directory of the shell subprocess will be the directory which the contents of the current buffer was read from.

This command won't work on Amigas!

Each `*shell*` buffer installs the major mode `shell-mode`. This provides the following commands.

`Ctrl-a`

Move the cursor to the beginning of the current line, *after* the prompt which the shell printed (if one exists).

`Ctrl-d`

If the cursor is at the end of the buffer send the shell process the `eof` character (`^D`) (signifying the end of the file). Otherwise delete the character under the cursor.

`RET`

Send the current line to the shell (minus any prompt at the beginning of the line). If the cursor is not on the last line of the buffer (i.e. the most recent prompt) the current line is copied to the end of the buffer before being sent.

`Ctrl-c Ctrl-n`

Move the cursor to the next prompt in the buffer.

`Ctrl-c Ctrl-p`

Move to the previous prompt.

`Ctrl-c Ctrl-c`

Send the `intr` character (`^C`) to the shell process.

`Ctrl-c Ctrl-d`

Send the `eof` character (`^D`) to the shell.

`Ctrl-c Ctrl-z`

Send the `susp` character (`^Z`) to the shell.

`Ctrl-c Ctrl-\`

Send the `quit` character (`^^`) to the shell.

- Hook: `shell-mode-hook`

This hook is evaluated by the Shell mode after it has initialised itself (and started its subprocess).

The following variables customise the actions of the Shell mode.

- Variable: `shell-file-name`

This variable defines the file name of the shell to run. Its default value is either the value of the environment variable `SHELL` or if that doesn't exist the file `/bin/sh`.

- Variable: `shell-whole-line`

When this variable's value is non-`nil` the `RET` command always sends the whole of the current line (minus any prompt) even when the cursor is not at the end of the line. Otherwise only the part of the line before the cursor is sent.

The default value of this variable is `t`.

- Variable: shell-prompt-regexp
This buffer-local variable defines the regular expression used to match the prompt printed by the shell each time it waits for you to enter a shell command. By default it has the value `^[^#%>)]*[]#%>)] *` but this may be incorrect if you have modified your shell's prompt.

1.85 jade.guide/Simple Customisation

Simple Customisation

=====

The best way to tailor the editor to your own requirements is with your personal startup file. This is called `.jaderc` in your home directory (1), it is a file of Lisp forms evaluated when Jade initialises itself.

Usually, setting the values of variables in your startup file is enough to configure Jade how you want, the Lisp function to set a variable is called `setq`, it's first argument is the name of the variable, it's second the value you wish to set it to. This value will usually be one of the following data types,

```
"xyz"
```

A string `'xyz'`.

```
'123'
```

```
'0173'
```

```
'0x7b'
```

A number, all of the above have the value 123 (in decimal, octal and hexadecimal).

```
'nil'
```

```
't'
```

A boolean value, `'nil'` means false, or not true. `'t'` is the opposite (in fact, any value not `'nil'` is true).

My `.jaderc` file looks something like this (note that semicolons introduce comments),

```
;;; .jaderc -*-Lisp-*-
```

```
;; Size of tabs for Lisp source is 2
(setq lisp-body-indent 2)
```

```
;; When on an Amiga, flag that I don't want pull down menus
(when (amiga-p)
  (setq amiga-no-menus t))
```

```
;; When editing English-text use auto-filling
(add-hook 'text-mode-hook 'fill-mode-on)
```

```
;; -with a maximum of 74 characters in a line
```

```
(setq fill-column 74)
```

```
;; Start the edit server
(server-open)
```

Most simple customisations can be achieved by simply giving a variable a new value. Use the 'setq' special form to do this (a special form is a type of function) as in the examples above. If you wish to set variables interactively use the 'set' command:

```
'Meta-x set RET VARIABLE-NAME RET NEW-VALUE RET'.
```

The 'add-hook' function adds a function (in this case 'fill-mode-on') to be called when the specified hook (in this case 'text-mode-hook') is evaluated. The single-quote before the names means that the names are passed as constants; *not* their values. If you don't quite understand what I'm talking about don't worry.

For full documentation of Jade's programming language see

Programming Jade

.

----- Footnotes -----

(1) On the Amiga, your home directory is defined as the contents of the environment variable 'HOME'.

1.86 jade.guide/Programming Jade

Programming Jade

This chapter of the manual is a full guide to Jade's Lisp programming language, including documentation for most of the built-in functions.

Intro

Introduction and Lisp conventions

Data Types

Data types and values in Lisp

Numbers

Integers and arithmetic functions

Sequences

Ordered sequences of data values

Symbols

Symbols are uniquely named objects

Evaluation	Evaluating expressions
Control Structures	Special forms. Conditionals, loops, etc...
Variables	Symbols represent named variables
Functions	Functions are the building blocks of Lisp programs
Macros	User-defined control structures
Streams	Data sinks and sources; character streams
Loading	Programs are stored in files
Compiled Lisp	Making programs run faster
Hooks	Hooks allow the extending of Jade
Buffers	Buffers allow editing of files
Windows	Windows receive input and display buffers
Positions	Coordinates in buffers and cursor movement
Marks	Marks represent the position of a ↔ character in a file
Glyph Tables	Controlling the glyphs rendered for each ASCII character
Input Events	Objects which represent input events
Keymaps	Mappings between events and commands

Event Loop	The event loop reads input events and invokes commands
Editing Files	Files are edited in buffers
Text	Functions to edit buffers with
Writing Modes	Creating new editing modes
Prompting	Interactively asking the user a question
Files	Manipulating files in the filing system
Processes	Jade can launch and control subprocesses when running under Unix
Miscellaneous Functions	Functions which don't fit elsewhere in this manual
Debugging	How to debug Lisp programs
Tips	General recommendations for Lisp programmers ↔

1.87 jade.guide/Intro

Introduction

=====

As you have probably gathered by now, Jade is largely controlled by its built in programming language: a dialect of Lisp containing many extensions (non-standard data types and functions) to make it suitable for controlling an editor. Through this language Jade can be customised and extended.

I have attempted to make the "standard" portion of the language (i.e. anything a normal Lisp would have; not related to editing) as compatible with GNU Emacs Lisp as possible. In some areas this rule doesn't apply, there will usually be a good reason for this. A few functions have been

inspired by Common Lisp.

The areas of the language which control the `*editor*` are `*not*` compatible with Emacs; some functions may be similar but since the two editors are fundamentally different I have not attempted to conform with the Emacs API.

All programs written using only the information in this manual should be compatible with future revisions of Jade.

This following sections explain some of the most important Lisp concepts and the conventions I've used in this manual.

nil and t	Boolean values in Lisp
The Lisp Reader	Basic program structure
Notation	Special glyphs used
Descriptions	How functions and variables are documented

1.88 jade.guide/nil and t

nil and t

The two boolean values in Lisp are the symbols `'nil'` (FALSE) and `'t'` (TRUE). Both these symbols always evaluate to themselves (so they do not have to be quoted), any attempt to change their values is an error.

All of the conditional instructions regard `*anything*` which is not `'nil'` as being TRUE (i.e. not-FALSE). The actual symbol `'t'` should be used where a TRUE boolean value must be explicitly stated to increase the clarity of the code.

This is not the end of the story; `'nil'` actually has another meaning: it represents the empty list. This is a consequence of how lists are constructed in Lisp, a list of zero elements is stored as the symbol `'nil'`.

To the Lisp system itself there is absolutely no difference between `'()` (the notation for a list with zero elements) and `'nil'` (the symbol nil). When writing code however, the list notation is usually used when the programmer regards the value as a list and the `'nil'` notation when its value as a boolean is to be emphasised.

1.89 jade.guide/The Lisp Reader

The Lisp Reader

Lisp programs and functions are stored internally as normal Lisp data objects, the Lisp Reader is the process used to translate textual descriptions of Lisp objects into the data structures used to represent the objects.

The Lisp Reader is the collection of internal functions accessed by the 'read' Lisp function. It reads a character at a time from an input stream until it has parsed a whole Lisp object.

See

Data Types

.

1.90 jade.guide/Notation

Notation

Wherever an example of evaluating a Lisp form is shown it will be formatted like this,

```
(+ 1 2)
=> 3
```

The glyph '=>' is used to show the computed value of a form.

When two forms are shown as being exactly equivalent to one another the glyph '==' is used, for example,

```
(car some-variable) == (nth 0 some-variable)
```

Evaluating some forms result in an error being signalled, this is denoted by the 'error-->' glyph.

```
(read-file "/tmp/foo")
error--> File error: No such file or directory, /tmp/foo
```

1.91 jade.guide/Descriptions

Descriptions

The simplest type of descriptions are the descriptions of variables (see

Variables
) , they look something like,

- Variable: grains-of-sand
This imaginary variable contains the number of grains of sand in a one-mile long stretch of an averagely sandy beach.

Hooks (see

Hooks
) are also described in this format, the only

difference is that 'Variable:' is replaced by 'Hook:'.

Functions (see

Functions
) and macros (see
Macros
) have more complex

descriptions; as well as the name of the thing being described, they also have a list of arguments which the thing will accept. Each argument in the list is named and may be referred to in the body of the description.

Two 'special' arguments may be used, '&optional' and '&rest'. They have the same meaning as when used in the lambda-list of a function definition (see

Lambda Expressions
) , that is '&optional' means that all

further arguments are optional, and '&rest' means that zero or more argument values are coalesced into a list to be used as the value of the following argument.

An example function definition follows.

- Function: useless-function FIRST &optional SECOND &rest TAIL
This function returns a list consisting of the values SECOND (when undefined the number 42 is used), all the items in the list TAIL and FIRST.

```
(useless-function 'foo 'bar 'xyz 20)
=> (bar xyz 20 foo)
```

```
(useless-function '50)
=> (42 50)
```

Macros and commands (see

Commands
) are defined in the same way with

'Macro:' or 'Command:' replacing 'Function:'.

Special forms (see

Special Forms
) are described similarly to

functions except that the argument list is formatted differently since special forms are, by definition, more flexible in how they treat their arguments. Optional values are enclosed in square brackets ('[OPTIONAL-ARG]') and three dots ('REPEATED-ARG...') indicate where zero or more arguments are allowed.

1.92 jade.guide/Data Types

Data Types

=====

The way that data values are represented in Lisp is fundamentally different to more "conventional" languages such as C or Pascal: in Lisp each piece of data (a "Lisp Object") has two basic attributes, the actual data and a tag value defining the **type** of the object. This means that type checking is performed on the actual data itself, not on the "variable" holding the data.

All Lisp objects are a member of one of the primitive types; these are types built into the Lisp system and can represent things like strings, integers, cons cells, vectors, etc...

More complex types of object can be constructed from these primitive types, for example a vector of three elements could be regarded as a type 'triple' if necessary. In general, each separate type provides a predicate function which returns 't' when applied to an object of its type.

Types Summary

List of the most common types

Read Syntax

Some types can be constructed from source code

Printed Representation

All types can be printed

Equality Predicates

How to test two objects for equality

Comparison Predicates

Comparing two objects as scalars

Type Predicates

Each type has a predicate defining it

Garbage Collection

Reusing memory from stale objects

1.93 jade.guide/Types Summary

Types Summary

Each separate data type is documented in its own section, this is a just a table of the more common types.

"Integer"

32-bit signed integers. See
Numbers

.

"Cons cell"

An object containing two other Lisp objects. See
Cons Cells

.

"List"

A sequence of objects, in Lisp lists are not primitive types, instead they are made by chaining together Cons cells. See
Lists

.

"Vector"

A one-dimensional array of objects. See
Vectors

.

"String"

A vector of characters. See
Strings

.

"Array"

An ordered sequence of objects which can be accessed in constant time, either a vector or a string. See
Sequences

.

"Sequence"

An ordered sequence of objects, either a list or an array. See

Sequences

.

"Symbol"

A symbol is a named object; they are used to provide named variables and functions. See
Symbols

.

"File"

A link to a file in the operating system's filing system, allows access to the file as a stream. See

Files

.

"Stream"

Serial data sinks and sources. See
Streams

.

"Void"

No type, only used in symbols to represent an unset function or
variable value.

"Buffer"

A "space" in which text can be edited, buffers may be displayed in
a window and hence edited by the user. See
Buffers

.

"Window"

A physical window in the underlying window-system, used for input
and output.

"Position"

A pair of integers, used to represent the coordinates of a
character in a buffer. See
Positions

.

"Mark"

A position in a specified file, this file may either be a buffer
in memory or a named file. See
Marks

.

"Process"

An object through which processes may be created and controlled.
See

Processes

.

"Glyph Table"

A lookup-table which is used to map characters in a buffer to the
sequence of glyphs they are rendered as. See
Glyph Tables

.

"Keymap"

A set of key-sequence-to-command mappings; when installed in a
buffer it controls how the editor reacts to all input from the
user. See

Keymaps

.

"Event"

An (input-) event from a window.

1.94 jade.guide/Read Syntax

Read Syntax

As previously noted the Lisp reader translates textual descriptions of Lisp objects into the object they describe (source files are simply descriptions of objects). However, not all data types can be created in this way: in fact the only types which can are integers, strings, symbols, cons cells (or lists) and vectors, all others have to be created by calling functions.

Note that comments in a Lisp program are introduced by the semi-colon character (`;`). Whenever the Lisp reader encounters a semi-colon where it's looking for the read syntax of a new Lisp object it will discard the rest of the line of input. See

Comment Styles

.

The "read syntax" of an object is the string which when given to the reader as input will produce the object. The read syntax of each type of object is documented in that type's main section of this manual but here is a small taste of how to write each type.

Integers

An integer is simply the number written in either decimal, octal (when the number is preceded by `'0'`) or hexadecimal (when the number is preceded by `'0x'`). An optional minus sign may be the first character in a number. Some examples are,

```
42
=> 42

0177
=> 127

0xff
=> 255

-0x10
=> -16
```

Strings

The read syntax of a string is simply the string with a double-quote character (`'"`) at each end, for more details see

Strings

.

```
"This is a string"
```

Cons cells

A cons cell is written in what is known as "dotted pair notation" and is just the two objects in the cell separated by a dot and the whole thing in parentheses,

```
(CAR . CDR)
```

Lists

The syntax of a list is similar to a cons cell (since this is what lists are made of): no dot is used and there may be zero or more objects,

```
(OBJECT1 OBJECT2 OBJECT3 ...)
```

```
("foo" ("bar" "baz") 100)
```

The second example is a list of three elements, a string, another list and a number.

Vectors

The read syntax of a vector is very similar to that of a list, simply use square brackets instead of parentheses,

```
[OBJECT1 OBJECT2 OBJECT3 ...]
```

Symbols

A symbol's read syntax is simply its name, for example the read syntax of a symbol called 'my-symbol' is,

```
my-symbol
```

1.95 jade.guide/Printed Representation

```
Printed Representation
```

The "printed representation" of an object is the string produced when the object is printed (with one of the 'print' functions), this will usually be very similar to the read syntax of the object (see

```
Read Syntax
).
```

Objects which do not have a read syntax *do* have a printed representation, it will normally be of the form,

```
#<relevant text>
```

where the "relevant text" is object-dependent and usually describes the object and its contents. The reader will signal an error if it encounters a description of an object in the format '#<...>'.

1.96 jade.guide/Equality Predicates

Equality Predicates

- Function: eq ARG1 ARG2

Returns 't' when ARG1 and ARG2 are the same object. Two objects are the same object when they occupy the same place in memory and hence modifying one object would alter the other. The following Lisp fragments may illustrate this,

```
(eq "foo" "foo") ;the objects are distinct
=> nil
```

```
(eq t t) ;the same object -- the symbol 't'
=> t
```

Note that the result of 'eq' is undefined when called on two integer objects with the same value, see 'eql'.

- Function: equal ARG1 ARG2

The function 'equal' compares the structure of the two objects ARG1 and ARG2. If they are considered to be equivalent then 't' is returned, otherwise 'nil' is returned.

```
(equal "foo" "foo")
=> t
```

```
(equal 42 42)
=> t
```

```
(equal 42 0)
=> nil
```

```
(equal '(x . y) '(x . y))
=> t
```

- Function: eql ARG1 ARG2

This function is a cross between 'eq' and 'equal': if ARG1 and ARG2 are both numbers then the value of these numbers are compared. Otherwise it behaves in exactly the same manner as 'eq' does.

```
(eql 3 3)
=> t
```

```
(eql 1 2)
=> nil
```

```
(eql "foo" "foo")
=> nil
```

```
(eql 'x 'x)
=> t
```

1.97 jade.guide/Comparison Predicates

Comparison Predicates

These functions compare their two arguments in a scalar fashion, the arguments may be of any type but the results are only meaningful for numbers, strings (ASCII values of each byte compared until a non-matching pair is found then those two values are compared as numbers) and positions.

- Function: > ARG1 ARG2
Returns 't' when ARG1 is 'greater than' ARG2.
- Function: >= ARG1 ARG2
Returns 't' when ARG1 is 'greater than or equal to' ARG2.
- Function: < ARG1 ARG2
Returns 't' when ARG1 is 'less than' ARG2.
- Function: <= ARG1 ARG2
Returns 't' when ARG1 is 'less than or equal to' ARG2.

1.98 jade.guide/Type Predicates

Type Predicates

Each type has a corresponding predicate which defines the objects which are members of that type.

- * 'integerp'
 - * 'numberp'
 - * 'null'
 - * 'consp'
 - * 'listp'
 - * 'vectorp'
 - * 'subrp'
 - * 'functionp'
 - * 'sequencep'
 - * 'stringp'
 - * 'symbolp'
-

- * `posp`
- * `bufferp`
- * `windowp`
- * `markp`
- * `processp`
- * `filep`
- * `keymapp`
- * `eventp`
- * `commandp`

The documentation for these functions is with the documentation for the relevant type.

1.99 jade.guide/Garbage Collection

Garbage Collection

In Lisp, data objects are used very freely; a side effect of this is that it is not possible to (easily) know when an object is "stale", that is, no references to it exist and it can therefore be reused.

The "garbage collector" is used to overcome this problem; whenever enough new data objects have been allocated to make it worthwhile, everything stops and the garbage collector works its way through memory deciding which objects are still in use and which are stale. The stale objects are then recorded as being available for reuse and evaluation continues again.

- Function: `garbage-collect`
Runs the garbage collector, usually this function doesn't need to be called manually.
- Variable: `garbage-threshold`
The number of bytes of data which must be allocated before evaluation will pause and the garbage collector called.

Its default value is about 100K.

See

Idle Actions

.

1.100 jade.guide/Numbers

Numbers

=====

Currently Jade is only capable of representing integers, for this it uses signed 32-bit integers: this gives a range of -2147483648 through 0 to 2147483647.

The read syntax of an integer is simply the number written in decimal, octal or hexadecimal. If the integer starts with the string '0x' it is assumed to be hexadecimal or if it starts with a zero it is treated as octal. The first character may be an optional minus or plus sign (this should come before any base-specifier). Examples of valid integer read syntaxes for the number 42 could be '42', '0x2a', '052', '+052', ...

An integer's printed representation is simply the number printed in decimal with a preceding minus sign if it is negative.

- Function: `numberp OBJECT`
This function returns 't' if OBJECT is a number.
- Function: `integerp OBJECT`
This function returns 't' when OBJECT is an integer.

Arithmetic Functions

Adding and subtracting...

Bitwise Functions

Using integers as bit-sequences

Numeric Predicates

Comparing numbers

Characters

Integers are used to represent characters

1.101 jade.guide/Arithmetic Functions

Arithmetic Functions

=====

There are a number of functions which perform arithmetic operations on numbers, they take a varying number of integer objects as their arguments then return a new integer object as their result.

Note that none of these functions check for overflow.

- Function: `+ NUMBER1 &rest NUMBERS`

This functions adds its arguments then returns their sum.

- Function: - NUMBER1 &rest NUMBERS

If this function is just given one argument (NUMBER1) that number is negated and returned. Otherwise each of NUMBERS is subtracted from a running total starting with the value of NUMBER1.

```
(- 20)
=> -20
```

```
(- 20 10 5)
=> 5
```

- Function: * NUMBER1 &rest NUMBERS

This function multiplies its arguments then returns the result.

- Function: / NUMBER1 &rest NUMBERS

This function performs division, a running-total (initialised from NUMBER1 is successively divided by each of NUMBERS then the result is returned.

```
(/ 100 2)
=> 50
```

```
(/ 200 2 5)
=> 20
```

- Function: % DIVIDEND DIVISOR

Returns the remainder from dividing DIVIDEND by DIVISOR.

```
(mod 5 3)
=> 2
```

- Function: 1+ NUMBER

This function returns the result of adding one to NUMBER.

```
(1+ 42)
=> 43
```

- Function: 1- NUMBER

Returns NUMBER minus one.

1.102 jade.guide/Bitwise Functions

Bitwise Functions

=====

These functions operate on the bit string which an integer is made of.

- Function: lsh NUMBER COUNT

This function bit-shifts the integer NUMBER COUNT bits to the left, if COUNT is negative NUMBER is shifted to the right instead.

```
(lsh 1 8)
=> 256
```

```
(lsh 256 -8)
=> 1
```

- Function: ash NUMBER COUNT

Similar to 'lsh' except that an arithmetical shift is done, this means that the sign of NUMBER is always preserved.

```
(ash 1 8)
=> 256
```

```
(ash -1 2)
=> -4
```

- Function: logand NUMBER1 &rest NUMBERS

This function uses a bit-wise logical 'and' operation to combine all its arguments (there must be at least one argument).

```
(logand 15 8)
=> 8
```

```
(logand 15 7 20)
=> 4
```

- Function: logior NUMBER1 &rest NUMBERS

Uses a bit-wise logical 'inclusive-or' to combine all its arguments (there must always be at least one argument).

```
(logior 1 2 4)
=> 7
```

- Function: logxor NUMBER1 &rest NUMBERS

Uses a bitwise logical 'exclusive-or' to combine all its arguments (there must be at least one).

```
(logxor 7 3)
=> 4
```

- Function: lognot NUMBER

This function inverts all the bits in NUMBER.

```
(lognot 0)
=> -1
```

```
(lognot 2)
=> -3
```

```
(lognot -1)
=> 0
```

1.103 jade.guide/Numeric Predicates

Numeric Predicates

=====

For the documentation of the functions '>', '<', '>=' and '<=' see

Comparison Predicates

.

- Function: = NUMBER1 NUMBER2

This function returns 't' if the two integers NUMBER1 and NUMBER2 have the same value.

```
(= 1 1)
=> t
```

```
(= 1 0)
=> nil
```

- Function: /= NUMBER1 NUMBER2

This function will return 't' if NUMBER1 and NUMBER2 are not equal to each other.

```
(/= 1 1)
=> nil
```

```
(/= 1 0)
=> t
```

- Function: zerop NUMBER

Returns 't' if NUMBER is equal to zero.

1.104 jade.guide/Characters

Characters

In Jade characters are stored in integers. Their read syntax is a question mark followed by the character itself which may be an escape sequence introduced by a backslash. For details of the available escape sequences see

Strings

.

```
?a
=> 97
```

```
?\n
=> 10
```

```
?\177
=> 127
```

- Function: `alpha-char-p CHARACTER`
This function returns `'t'` when `CHARACTER` is one of the alphabetic characters.

```
(alpha-char-p ?a)
=> t
```

- Function: `upper-case-p CHARACTER`
When `CHARACTER` is one of the upper-case characters this function returns `'t'`.
- Function: `lower-case-p CHARACTER`
Returns `'t'` when `CHARACTER` is lower-case.
- Function: `digit-char-p CHARACTER`
This function returns `'t'` when `CHARACTER` is one of the decimal digit characters.
- Function: `alphanumericp CHARACTER`
This function returns `'t'` when `CHARACTER` is either an alphabetic character or a decimal digit character.
- Function: `space-char-p CHARACTER`
Returns `'t'` when `CHARACTER` is a white-space character (space, tab, newline or form feed).
- Function: `char-upcase CHARACTER`
This function returns the upper-case equivalent of `CHARACTER`. If `CHARACTER` is already upper-case or has no upper-case equivalent it is returned unchanged.

```
(char-upcase ?a)
=> 65 ; 'A'
```

```
(char-upcase ?A)
=> 65 ; 'A'
```

```
(char-upcase ?!)
=> 33 ; '!'
```

- Function: `char-downcase CHARACTER`
Returns the lower-case equivalent of the character `CHARACTER`.

1.105 jade.guide/Sequences

Sequences

=====

Sequences are ordered groups of objects, there are several primitive types which can be considered sequences, each with its own good and bad points.

A sequence is either an array or a list, where an array is either a

vector or a string.

- Function: sequencep OBJECT
This function returns 't' if OBJECT is a sequence, 'nil' otherwise.

Cons Cells	An ordered pair of two objects
Lists	Chains of cons cells
Vectors	A chunk of memory holding a number of objects ↔
Strings	Strings are efficiently-stored vectors
Array Functions	Accessing elements in vectors and strings
Sequence Functions	These work on any type of sequence

1.106 jade.guide/Cons Cells

Cons Cells

A "cons cell" is an ordered pair of two objects, the "car" and the "cdr".

The read syntax of a cons cell is an opening parenthesis followed by the read syntax of the car, a dot, the read syntax of the cdr and a closing parenthesis. For example a cons cell with a car of 10 and a cdr of the string 'foo' would be written as,

```
(10 . "foo")
```

- Function: cons CAR CDR
This function creates a new cons cell. It will have a car of CAR and a cdr of CDR.

```
(cons 10 "foo")
=> (10 . "foo")
```

- Function: consp OBJECT
This function returns 't' if OBJECT is a cons cell and 'nil' otherwise.

```
(consp '(1 . 2))
```

```
=> t

(cons nil)
=> nil

(cons (cons 1 2))
=> t
```

In Lisp an "atom" is any object which is not a cons cell (and is, therefore, atomic).

- Function: atom OBJECT
Returns 't' if OBJECT is an atom (not a cons cell).

Given a cons cell there are a number of operations which can be performed on it.

- Function: car CONS-CELL
This function returns the object which the car of the cons cell CONS-CELL.

```
(car (cons 1 2))
=> 1

(car '(1 . 2))
=> 1
```

- Function: cdr CONS-CELL
This function returns the cdr of the cons cell CONS-CELL.

```
(cdr (cons 1 2))
=> 2

(cdr '(1 . 2))
=> 2
```

- Function: rplaca CONS-CELL NEW-CAR
This function sets the value of the car in the cons cell CONS-CELL to NEW-CAR. The value returned is NEW-CAR.

```
(setq x (cons 1 2))
=> (1 . 2)
(rplaca x 3)
=> 3
x
=> (3 . 2)
```

- Function: rplacd CONS-CELL NEW-CDR
This function is similar to 'rplacd' except that the cdr slot of CONS-CELL is modified.

1.107 jade.guide/Lists

Lists

A list is a sequence of zero or more objects, the main difference between lists and vectors is that lists are more dynamic: they can change size, be split, reversed, concatenated, etc... very easily.

In Lisp lists are not a primitive type; instead singly-linked lists are created by chaining cons cells together (see Cons Cells).

- Function: listp OBJECT
This functions returns 't' when its argument, OBJECT, is a list (i.e. either a cons cell or 'nil').

List Structure

How lists are built from cons cells

Building Lists

Dynamically creating lists

Accessing List Elements

Getting at the elements which make the list

Modifying Lists

How to alter the contents of a list

Association Lists

Lists can represent relations

Infinite Lists

Circular data structures in Lisp

1.108 jade.guide/List Structure

List Structure

.....

Each element in a list is given its own cons cell and stored in the car of that cell. The list object is then constructed by making the cdr of a cell contain the cons cell of the next element (and hence the whole tail of the list). The cdr of the cell containing the last element in the list is 'nil'. A list of zero elements is represented by the symbol 'nil'.

The read syntax of a list is an opening parenthesis, followed by the read syntax of zero or more space-separated objects, followed by a closing parenthesis. Alternatively, lists can be constructed 'manually' using dotted-pair notation.

All of the following examples result in the same list of five elements: the numbers from zero to four.

```
(0 1 2 3 4)
```

```
(0 . (1 . (2 . (3 . (4 . nil)))))
```

```
(0 1 2 . (3 4))
```

An easy way to visualise lists and how they are constructed is to see each cons cell in the list as a separate "box" with pointers to its car and cdr,

```
+-----+-----+
|  o  |  o----> cdr
+---|---+-----+
|
--> car
```

Complex box-diagrams can now be drawn to represent lists. For example the following diagram represents the list '(1 2 3 4)'.

```
+-----+-----+ +-----+-----+ +-----+-----+ +-----+-----+
|  o  |  o----> |  o  |  o----> |  o  |  o----> |  o  |  o----> nil
+---|---+-----+ +---|---+-----+ +---|---+-----+ +---|---+-----+
|                   |                   |                   |
--> 1                 --> 2                 --> 3                 --> 4
```

A more complex example, the list '((1 2) (foo bar))' can be drawn as,

```
+-----+-----+                                     +-----+-----+
|  o  |  o----->                                     |  o  |  o----> nil
+---|---+-----+                                     +---|---+-----+
|
+-----+-----+ +-----+-----+                 +-----+-----+ +-----+-----+
|  o  |  o----> |  o  |  o----> nil                 |  o  |  o----> |  o  |  o----> nil
+---|---+-----+ +---|---+-----+                 +---|---+-----+ +---|---+-----+
|                   |                   |                   |                   |
--> 1                 --> 2                 --> foo                 --> bar
```

Sometimes when manipulating complex list structures it is very helpful to make a diagram of what it is that's being manipulated.

1.109 jade.guide/Building Lists

Building Lists

.....

It has already been shown how you can create lists using the Lisp reader; this method does have a drawback though: the list created is effectively static. If you modify the contents of the list and that list was created when a function was defined the list will remain modified for all future invocations of that function. This is not usually a good idea, consider the following function definition,

```
(defun bogus-function (x)
  "Return a list whose first element is nil and whose second element is X."
  (let
    ((result '(nil nil)))      ;Static list which is filled in each time
    (rplaca (cdr result) x)    ; the function is called
    result))
```

This function does in fact do what its documentation claims, but a problem arises when it is called more than once,

```
(setq x (bogus-function 'foo))
=> (nil foo)
(setq y (bogus-function 'bar))
=> (nil bar)                ;The first result has been destroyed
x
=> (nil bar)                ;See!
```

This example is totally contrived -- no one would ever write a function like the one in the example but it nicely demonstrates the need for a dynamic method of creating lists.

- Function: list &rest ELEMENTS

This function creates a list out of its arguments, if zero arguments are given the empty list, 'nil', is returned.

```
(list 1 2 3)
=> (1 2 3)

(list (major-version-number) (minor-version-number))
=> (3 2)

(list)
=> nil                ;Equivalent to '()'
```

- Function: make-list LENGTH &optional INITIAL-VALUE

This function creates a list LENGTH elements long. If the INITIAL-VALUE argument is given it defines the value of all elements in the list, if it is not given they are all 'nil'.

```
(make-list 2)
=> (nil nil)

(make-list 3 t)
=> (t t t)

(make-list 0)
=> nil
```

- Function: append &rest LISTS

This function creates a new list with the elements of each of its arguments (which must be lists). Unlike the function 'nconc' this function preserves all of its arguments.

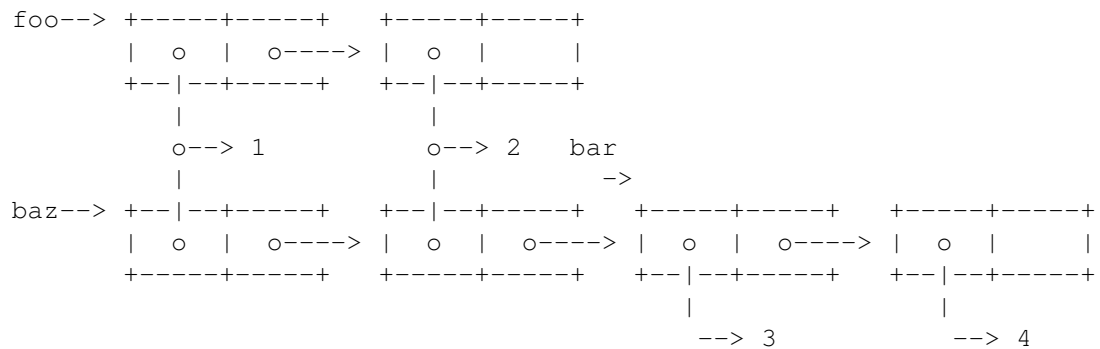
```
(append '(1 2 3) '(4 5))
=> (1 2 3 4 5)
```

```
(append)
=> nil
```

What actually happens is that all arguments but the last are copied then the last argument is linked on to the end of the list (uncopied).

```
(setq foo '(1 2))
=> (1 2)
(setq bar '(3 4))
=> (3 4)
(setq baz (append foo bar))
=> (1 2 3 4)
(eq (nthcdr 2 baz) bar)
=> t
```

The following diagram shows the final state of the three variables more clearly,



Note how 'foo' and the first half of 'baz' use the *same* objects for their elements -- copying a list only copies its cons cells, its elements are reused. Also note how the variable 'bar' actually references the mid-point of 'baz' since the last list in an 'append' call is not copied.

- Function: reverse LIST

This function returns a new list; it is made from the elements of the list LIST in reverse order. Note that this function does not alter its argument.

```
(reverse '(1 2 3 4))
=> (4 3 2 1)
```

As a postscript to this section, the function used as an example at the beginning could now be written as,

```
(defun not-so-bogus-function (x)
  (list nil x))
```

Also note that the 'cons' function can be used to create lists by hand and to add new elements onto the front of a list.

1.110 jade.guide/Accessing List Elements

Accessing List Elements

.....

The most powerful method of accessing an element in a list is via a combination of the 'car' and 'cdr' functions. There are other functions which provide an easier way to get at the elements in a flat list. These will usually be faster than a string of 'car' and 'cdr' operations.

- Function: nth COUNT LIST

This function returns the element COUNT elements down the list, therefore to access the first element use a COUNT of zero (or even better the 'car' function). If there are too few elements in the list and no element number COUNT can be found 'nil' is returned.

```
(nth 3 '(0 1 2 3 4 5))  
=> 3
```

```
(nth 0 '(foo bar))  
=> foo
```

- Function: nthcdr COUNT LIST

This function takes the cdr of the list LIST COUNT times, returning the last cdr taken.

```
(nthcdr 3 '(0 1 2 3 4 5))  
=> (3 4 5)
```

```
(nthcdr 0 '(foo bar))  
=> (foo bar)
```

- Function: last LIST

This function returns the last element in the list LIST. If the list has zero elements 'nil' is returned.

```
(last '(1 2 3))  
=> 3
```

```
(last '())  
=> nil
```

- Function: member OBJECT LIST

This function scans through the list LIST until it finds an element which is 'equal' to OBJECT. The tail of the list (the cons cell whose car is the matched object) is then returned. If no elements match OBJECT then the empty list 'nil' is returned.

```
(member 'c '(a b c d e))  
=> (c d e)
```

```
(member 20 '(1 2))  
=> nil
```

- Function: memq OBJECT LIST

This function is similar to 'member' except that comparisons are performed by the 'eq' function not 'equal'.

1.111 jade.guide/Modifying Lists

Modifying Lists

.....

The 'nthcdr' function can be used in conjunction with the 'rplaca' function to modify an arbitrary element in a list. For example,

```
(rplaca (nthcdr 2 '(0 1 2 3 4 5)) 'foo)
=> foo
```

sets the third element of the list '(0 1 2 3 4 5)' to the symbol called 'foo'.

There are also functions which modify the structure of a whole list. These are called "destructive" operations because they modify the actual structure of a list -- no copy is made. This can lead to unpleasant side effects if care is not taken.

- Function: nconc &rest LISTS

This function is the destructive equivalent of the function 'append', it modifies its arguments so that it can return a list which is the concatenation of the elements in its arguments lists.

Like all the destructive functions this means that the lists given as arguments are modified (specifically, the cdr of their last cons cell is made to point to the next list). This can be seen with the following example (similar to the example in the 'append' documentation).

```
(setq foo '(1 2))
=> (1 2)
(setq bar '(3 4))
=> (3 4)
(setq baz (nconc foo bar))
=> (1 2 3 4)
foo
=> (1 2 3 4)                ; 'foo' has been altered!
(eq (nthcdr 2 baz) bar)
=> t
```

The following diagram shows the final state of the three variables more clearly,

```
foo-->
baz--> +-----+-----+ +-----+-----+ +-----+-----+ +-----+-----+
      | o | | o----> | o | | o----> | o | | o----> | o | |
      +---|---+-----+ +---|---+-----+ +---|---+-----+ +---|---+-----+
          |               |               |               |
          --> 1           --> 2           --> 3           --> 4
```

- Function: `nreverse` LIST

This function rearranges the cons cells constituting the list LIST so that the elements are in the reverse order to what they were.

```
(setq foo '(1 2 3))
=> (1 2 3)
(nreverse foo)
=> (3 2 1)
foo
=> (1) ; 'foo' wasn't updated when the list
; was altered.
```

- Function: `delete` OBJECT LIST

This function destructively removes all elements of the list LIST which are 'equal' to OBJECT then returns the modified list.

```
(delete t '(nil t nil t nil))
=> (nil nil nil)
```

When this function is used to remove an element from a list which is stored in a variable that variable must be set to the return value of the 'delete' function. Otherwise, if the first element of the list has to be deleted (because it is 'equal' to OBJECT) the value of the variable will not change.

```
(setq foo '(1 2 3))
=> (1 2 3)
(delete 1 foo)
=> (2 3)
foo
=> (1 2 3)
(setq foo (delete 1 foo))
=> (2 3)
```

- Function: `delq` OBJECT LIST

This function is similar to the 'delete' function, the only difference is that the 'eq' function is used to compare OBJECT with each of the elements in LIST, instead of the 'equal' function which is used by 'delete'.

1.112 jade.guide/Association Lists

Association Lists

.....

An "association list" (or "alist") is a list mapping key values to other values. Each element of the alist is a cons cell, the car of which is the "key", the cdr is the value that it associates to. For example an alist could look like,

```
((fred . 20)
 (bill . 30))
```

this alist has two keys, 'fred' and 'bill' which both associate to an

integer (20 and 30 respectively).

It is possible to make the associated values lists, this looks like,

```
((fred 20 male)
 (bill 30 male)
 (sue 25 female))
```

in this alist the symbol 'fred' is associated with the list '(20 male)'.

There are a number of functions which let you interrogate an alist with a given key for its association.

- Function: `assoc` KEY ALIST

This function scans the association list ALIST for the first element whose car is 'equal' to KEY, this element is then returned. If no match of KEY is found 'nil' is returned.

```
(assoc 'two '((one . 1) (two . 2) (three . 3)))
=> (two . 2)
```

- Function: `assq` KEY ALIST

Similar to the function 'assoc' except that the function 'eq' is used to compare elements instead of 'equal'.

It is not usually wise to use 'assq' when the keys of the alist may not be symbols -- 'eq' won't think two objects are equivalent unless they are the *same* object!

```
(assq "foo" '(("bar" . 1) ("foo" . 2)))
=> nil
(assoc "foo" '(("bar" . 1) ("foo" . 2)))
=> ("foo" . 2)
```

- Function: `rassoc` ASSOCIATION ALIST

This function searches through ALIST until it finds an element whose cdr is 'equal' to ASSOCIATION, that element is then returned. 'nil' will be returned if no elements match.

```
(rassoc 2 '((one . 1) (two . 2) (three . 3)))
=> (two . 2)
```

- Function: `rassq` ASSOCIATION ALIST

This function is equivalent to 'rassoc' except that it uses 'eq' to make comparisons.

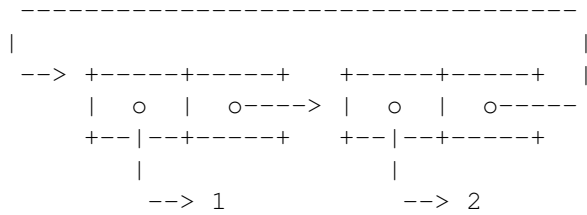
1.113 jade.guide/Infinite Lists

Infinite Lists

.....

Sometimes it is useful to be able to create 'infinite' lists -- that is, lists which appear to have no last element -- this can easily be done in Lisp by linking the cdr of the last cons cell in the list

structure back to the beginning of the list.



The diagram above represents the infinite list '(1 2 1 2 1 2 ...)'.
 Note: The diagram shows a circular structure where the end of one list points back to the beginning of another, forming a cycle.

Infinite lists have a major drawback though, many of the standard list manipulation functions can not be used on them. These functions work by moving through the list until they reach the end. If the list has *no* end the function may never terminate and the only option is to send Jade an interrupt signal (see `Interrupting Jade`).

The only functions which may be used on circular lists are: the cons cell primitives ('cons', 'car', 'cdr', 'rplaca', 'rplacd'), 'nth' and 'nthcdr'.

Also note that infinite lists can't be printed.

1.114 jade.guide/Vectors

Vectors

A vector is a fixed-size sequence of Lisp objects, each element may be accessed in constant time -- unlike lists where the time taken to access an element is proportional to the position of the element.

The read syntax of a vector is an opening square bracket, followed by zero or more space-separated objects, followed by a closing square bracket. For example,

```
[zero one two three]
```

In general it is best to use vectors when the number of elements to be stored is known and lists when the sequence must be more dynamic.

- Function: `vectorp OBJECT`
 This function returns 't' if its argument, OBJECT, is a vector.
- Function: `vector &rest ELEMENTS`
 This function creates a new vector containing the arguments given to the function.

```
(vector 1 2 3)
=> [1 2 3]
```

```
(vector)
=> []
```

- Function: `make-vector SIZE &optional INITIAL-VALUE`
Returns a new vector, `SIZE` elements big. If `INITIAL-VALUE` is defined each element of the new vector is set to `INITIAL-VALUE`, otherwise they are all `'nil'`.

```
(make-vector 4)
=> [nil nil nil nil]
```

```
(make-vector 2 t)
=> [t t]
```

1.115 jade.guide/Strings

Strings

A string is a vector of characters (see Characters), they are

generally used for storing and manipulating pieces of text. Jade puts no restrictions on the values which may be stored in a string -- specifically, the null character (`'^@'`) may be stored with no problems.

The read syntax of a string is a double quote character, followed by the contents of the string, the object is terminated by a second double quote character. For example, `"abc"` is the read syntax of the string `'abc'`.

Any backslash characters in the string's read syntax introduce an escape sequence; one or more of the following characters are treated specially to produce the next *actual* character in the string.

The following escape sequences are supported (all are shown without their leading backslash `'\'` character).

`'n'`
A newline character.

`'r'`
A carriage return character.

`'f'`
A form feed character.

`'t'`
A TAB character.

`'a'`
A 'bell' character (this is Ctrl-g).

``^C'`

The `'control'` code of the character C. This is calculated by toggling the seventh bit of the `*upper-case*` version of C.

For example,

```
\^C           ;A Ctrl-c character (ASCII value 3)
\^@           ;The NUL character (ASCII value 0)
```

``012'`

The character whose ASCII value is the octal value `'012'`. After the backslash character the Lisp reader reads up to three octal digits and combines them into one character.

``x12'`

The character whose ASCII value is the hexadecimal value `'12'`, i.e. an `'x'` character followed by one or two hex digits.

- Function: `stringp` OBJECT

This function returns `'t'` if its argument is a string.

- Function: `make-string` LENGTH &optional INITIAL-CHARACTER

Creates a new string containing LENGTH characters, each character is initialised to INITIAL-CHARACTER (or to spaces if INITIAL-CHARACTER is not defined).

```
(make-string 3)
=> "   "
```

```
(make-string 2 ?$)
=> "$ $"
```

- Function: `concat` &rest ARGS

This function concatenates all of its arguments, ARGS, into a single string which is returned. If no arguments are given then the null string (`'`) results.

Each of the ARGS may be a string, a character or a list or vector of characters. Characters are stored in strings modulo 256.

```
(concat "foo" "bar")
=> "foobar"
```

```
(concat "a" ?b)
=> "ab"
```

```
(concat "foo" [?b ?a ?r])
=> "foobar"
```

```
(concat)
=> ""
```

- Function: `substring` STRING START &optional END

This function creates a new string which is a partial copy of the string STRING. The first character copied is START characters from the beginning of the string. If the END argument is defined it is the index of the character to stop copying at, if it is not defined

all characters until the end of the string are copied.

```
(substring "xxyfoozwx" 3 6)
=> "foo"
```

```
(substring "xyzfoobar" 3)
=> "foobar"
```

- Function: `string= STRING1 STRING2`

This function compares the two strings `STRING1` and `STRING2` -- if they are made from the same characters in the same order then `'t'` is returned, else `'nil'`.

```
(string= "one" "one")
=> t
```

```
(string= "one" "two")
=> nil
```

Note that an alternate way to compare strings (or anything!) is to use the `'equal'` function.

- Function: `string< STRING1 STRING2`

This function returns `'t'` if `STRING1` is `'less'` than `'string2'`. This is determined by comparing the two strings a character at a time, the first pair of characters which do not match each other are then compared with a normal `'less-than'` function.

In Jade the standard `'<'` function understands strings so `'string<'` is just a macro calling that function.

```
(string< "abc" "abd")
=> t
```

```
(string< "abc" "abb")
=> nil
```

Functions are also available which match regular expressions with strings (see

`Search and Match Functions`

) and which apply a mapping to

each character in a string (see

`Translation Functions`

).

1.116 jade.guide/Array Functions

Array Functions

- Function: `arrayp OBJECT`

This function returns `'t'` if `OBJECT` is an array.

- Function: `aref ARRAY POSITION`
Returns the element of the array (vector or string) `ARRAY` `POSITION` elements from the first element (i.e. the first element is numbered zero). If no element exists at `POSITION` in `ARRAY`, `'nil'` is returned.

```
(aref [0 1 2 3] 2)
=> 2

(aref "abcdef" 3)
=> 100 ; 'd'
```

- Function: `aset ARRAY POSITION VALUE`
This function sets the element of the array `ARRAY` with an index of `POSITION` (counting from zero) to `VALUE`. An error is signalled if element `POSITION` does not exist. The result of the function is `VALUE`.

```
(setq x [0 1 2 3])
=> [0 1 2 3]
(aset x 2 'foo)
=> foo
x
=> [0 1 foo 3]
```

1.117 jade.guide/Sequence Functions

Sequence Functions

- Function: `length SEQUENCE`
This function returns the length (an integer) of the sequence `SEQUENCE`.

```
(length "abc")
=> 3

(length '(1 2 3 4))
=> 4

(length [x y])
=> 2
```

- Function: `copy-sequence SEQUENCE`
Returns a new copy of the sequence `SEQUENCE`. Where possible (in lists and vectors) only the 'structure' of the sequence is newly allocated: the same objects are used for the elements in both sequences.

```
(copy-sequence "xy")
=> "xy"

(setq x ('("one" "two")))
=> ("one" "two")
```

```
(setq y (copy-sequence x))
=> ("one" "two")
(eq x y)
=> nil
(eq (car x) (car y))
=> t
```

- Function: elt SEQUENCE POSITION

This function returns the element of SEQUENCE POSITION elements from the beginning of the sequence.

This function is a combination of the 'nth' and 'aref' functions.

```
(elt [0 1 2 3] 1)
=> 1
```

```
(elt '(foo bar) 0)
=> foo
```

1.118 jade.guide/Symbols

Symbols

=====

Symbols are objects with a name (usually a unique name), they are one of the most important data structures in Lisp since they are used to provided named variables (see

Variables
) and functions (see
Functions
).

- Function: symbolp OBJECT

This function returns 't' when its argument is a symbol.

Symbol Syntax

The read syntax of symbols

Symbol Attributes

The objects stored in a symbol

Obarrays

Vectors used to store symbols

Creating Symbols

Allocating new symbols

Interning

Putting a symbol into an obarray

Property Lists

Each symbol has a set of properties

1.119 jade.guide/Symbol Syntax

Symbol Syntax

The read syntax of a symbol is simply its name; if the name contains any meta-characters (whitespace or any from '()[]'";|') they will have to be entered specially. There are two ways to tell the reader that a meta-character is actually part of the symbol's name:

1. Precede the meta-character by a backslash character ('\'), for example:

```
xy\(z\)           ;the symbol whose name is 'xy(z)'
```

2. Enclose part of the name in vertical lines (two '|' characters). All characters after the starting vertical line are copied as-is until the closing vertical line is encountered. For example:

```
xy|(z)|           ;the symbol 'xy(z)'
```

Here are some example read syntaxes.

```
setq           ; 'setq'
|setq|        ; 'setq'
\s\e\t\q      ; 'setq'
1             ; the *number* 1
\1           ; the *symbol* '1'
|!$%zf78&|   ; '!$%zf78&'
foo|(bar)|    ; 'foo(bar)'
foo\(bar\)   ; 'foo(bar)'
```

1.120 jade.guide/Symbol Attributes

Symbol Attributes

All symbols have four basic attributes, most important is the "print name" of the symbol. This is a string containing the name of the symbol, after it has been defined (when the symbol is first created) it may not be changed.

- Function: symbol-name SYMBOL

This function returns the print name of the symbol SYMBOL.

```
(symbol-name 'unwind-protect)
=> "unwind-protect"
```

Each symbol also has a "value" cell storing the value of this symbol when it is referenced as a variable. Usually this cell is accessed implicitly by evaluating a variable form but it can also be read via the 'symbol-value' function(1) (see

Variables
)

Similar to the value cell each symbol also has a "function" cell which contains the function definition of the symbol (see

Named Functions
)

The 'symbol-function' function can be used to read this cell and the 'fset' function to set it.

Lastly, there is the symbol's "property list", this is similar to an alist (see

Association Lists
) and provides a method of storing arbitrary extra values in each symbol. See
Property Lists
.

----- Footnotes -----

(1) Actually buffer-local variables complicate matters but you'll learn about that later.

1.121 jade.guide/Obarrays

Obarrays

An "obarray" is the structure used to ensure that no two symbols have the same name and to provide quick access to a symbol given its name. An obarray is basically a vector (with a slight wrinkle), each element of the vector is a chain of symbols which share the same hash-value (a "bucket"). These symbols are chained together through links which are invisible to Lisp programs: if you examine an obarray you will see that each bucket looks as though it has at most one symbol stored in it.

The normal way to reference a symbol is simply to type its name in the program, when the Lisp reader encounters a name of a symbol it looks in the default obarray for a symbol of that name. If the named symbol doesn't exist it is created and hashed into the obarray -- this process is known as "interning" the symbol, for more details see

Interning
.

- Variable: obarray

This variable contains the obarray that the 'read' function uses when interning symbols. If you change this I hope you know what you're doing.

- Function: `make-obarray SIZE`
This function creates a new obarray with `SIZE` hash buckets (this should be a prime number for best results).

This is the only correct way of making an obarray.
- Function: `find-symbol SYMBOL-NAME &optional OBARRAY`
This function scans the specified obarray (`OBARRAY` or the value of the variable `'obarray'` if `OBARRAY` is undefined) for a symbol whose name is the string `SYMBOL-NAME`. The value returned is the symbol if it can be found or `'nil'` otherwise.


```
(find-symbol "setq")  
=> setq
```
- Function: `apropos REGEXP &optional PREDICATE OBARRAY`
Returns a list of symbols from the obarray `OBARRAY` (or the default) whose print name matches the regular expression `REGEXP`. If `PREDICATE` is defined and not `'nil'`, each symbol which matches `REGEXP` is applied to the function `PREDICATE`, if the value is `'t'` it is considered a match.

The `PREDICATE` argument is useful for restricting matches to a certain type of symbol, for example only commands.

```
(apropos "^yank" 'commandp)  
=> (yank-rectangle yank yank-to-mouse)
```

1.122 jade.guide/Creating Symbols

Creating Symbols

It is possible to allocate symbols dynamically, this is normally only necessary when the symbol is to be interned in the non-default obarray or the symbol is a temporary object which should not be interned (for example: labels in a compiler?).

- Function: `make-symbol PRINT-NAME`
This function creates and returns a new, uninterned, symbol whose print name is the string `PRINT-NAME`. Its variable and function value cells are void and it will have an empty property list.


```
(make-symbol "foo")  
=> foo
```
- Function: `gensym`
This function returns a new, uninterned, symbol which has a unique print name.

```
(gensym)  
=> G0001
```

```
(gensym)
=> G0002
```

1.123 jade.guide/Interning

Interning

"Interning" a symbol means to store it in an obarray so that it can be found in the future: all variables and named-functions are stored in interned symbols.

When a symbol is interned a hash function is applied to its print name to determine which bucket in the obarray it should be stored in. Then it is simply pushed onto the front of that bucket's chain of symbols.

Normally all interning is done automatically by the Lisp reader. When it encounters the name of a symbol which it can't find in the default obarray (the value of the variable 'obarray') it creates a new symbol of that name and interns it. This means that no two symbols can have the same print name, and that the read syntax of a particular symbol always produces the same object (unless the value of 'obarray' is altered).

```
(eq 'some-symbol 'some-symbol)
=> t
```

- Function: intern SYMBOL-NAME &optional OBARRAY

This function uses 'find-symbol' to search the OBARRAY (or the standard obarray) for a symbol called SYMBOL-NAME. If a symbol of that name is found it is returned, otherwise a new symbol of that name is created, interned into the obarray, and returned.

```
(intern "setq")
=> setq
```

```
(intern "my-symbol" my-obarray)
=> my-symbol
```

- Function: intern-symbol SYMBOL &optional OBARRAY

Interns the symbol SYMBOL into the obarray OBARRAY (or the standard one) then returns the symbol. If SYMBOL is currently interned in an obarray an error is signalled.

```
(intern-symbol (make-symbol "foo"))
=> foo
```

```
(intern-symbol 'foo)
error--> Error: Symbol is already interned, foo
```

- Function: unintern SYMBOL &optional OBARRAY

This function removes the symbol SYMBOL from the obarray OBARRAY then returns the symbol.

Beware! this function must be used with **extreme** caution -- once you unintern a symbol there's no way to recover it.

```
(unintern 'setq)           ;This is extremely stupid
=>setq
```

1.124 jade.guide/Property Lists

Property Lists

Each symbol has a property list (or "plist"), this is a structure which associates an arbitrary Lisp object with a key (usually a symbol). The keys in a plist may not have any duplications (so that each property is only defined once).

The concept of a property list is very similar to an association list (see

```
Association Lists
) but there are two main differences:
```

1. Structure; each element of an alist represents one key/association pair. In a plist each pair of elements represents an association: the first is the key, the second the property. For example, where an alist may be,

```
((one . 1) (two . 2) (three . 3))
```

a property list would be,

```
(one 1 two 2 three 3)
```

2. Plists have their own set of functions to modify the list. This is done destructively, altering the property list (since the plist is stored in only one location, the symbol, this is quite safe).

- Function: get SYMBOL PROPERTY

This function searches the property list of the symbol SYMBOL for a property 'eq' to PROPERTY. If such a property is found it is returned, else the value 'nil' is returned.

```
(get 'if 'lisp-indent)
=> 2
```

```
(get 'set 'lisp-indent)
=> nil
```

- Function: put SYMBOL PROPERTY NEW-VALUE

'put' sets the value of the property PROPERTY to NEW-VALUE in the property list of the symbol SYMBOL. If there is an existing value for this property it is overwritten. The value returned is NEW-VALUE.

```
(put 'foo 'prop 200)
=> 200
```

- Function: `symbol-plist SYMBOL`
Returns the property list of the symbol `SYMBOL`.

```
(symbol-plist 'if)
=> (lisp-indent 2)
```

- Function: `setplist SYMBOL PLIST`
This function sets the property list of the symbol `SYMBOL` to `PLIST`.

```
(setplist 'foo '(zombie yes))
=> (zombie yes)
```

1.125 jade.guide/Evaluation

Evaluation

=====

So far I have only discussed a few of the various data types available and how the Lisp reader can convert textual descriptions of these types into Lisp objects. Obviously there has to be a way of actually computing something -- it would be difficult to write a useful program otherwise.

What sets Lisp apart from other languages is that in Lisp there is no difference between programs and data: a Lisp program is just a sequence of Lisp objects which will be interpreted when the program is run.

The subsystem which does this interpreting is called the "Lisp evaluator" and each expression to be evaluated is called a "form". The evaluator (the function `'eval'`) examines the structure of the form that is applied to and computes the value of the form within the current environment.

A form can be any type of data object; the only types which the evaluator treats specially are symbols (which stand for variables) and lists, anything else is returned as-is (and is called a "self-evaluating form").

- Function: `eval FORM`
This function computes the value of the form which is its argument, within the current environment. The computed value is then returned. `'eval'` is the basic function for interpreting Lisp objects.

Symbol Forms

How variables are accessed

List Forms

Subroutine calls

Self-Evaluating Forms

Forms which don't get evaluated

Quoting

How to prevent evaluation of forms

1.126 jade.guide/Symbol Forms

Symbol Forms

When the evaluator is applied to a symbol the computed value of the form is the object stored in the symbol's variable slot. Basically this means that to get the value of a variable you simply write its name. For example,

```
buffer-list
=> (#<buffer *jade*> #<buffer programmer.texti>)
```

this extract from a Lisp session shows the read syntax of a form to get the value of the variable 'buffer-list' and the result when this form is evaluated.

Since forms are evaluated within the current environment the value of a variable is its newest binding, or in the case of buffer-local variables, its value in the current buffer. See

Variables

.

If the value of an evaluated symbol is void an error is signalled.

1.127 jade.guide/List Forms

List Forms

Forms which are lists are used to call a subroutine. The first element of the list is the subroutine which is to be called; all further elements are arguments to be applied to the subroutine.

There are several different types of subroutines available: functions, macros, special forms and autoloads. When the evaluator finds a form which is a list it tries to classify the form into one of these four types. First of all it looks at the first element of the list, if it is a symbol it gets the value from the function slot of the symbol (note that the first element of a list form is **never** evaluated

itself). This value (either the first element or the symbol's function value) is enough to classify the form into one of the four types.

```

Function Call Forms
    'Normal' subroutines

Macro Call Forms
    Source code expansions

Special Forms
    Abnormal control structures

Autoload Forms
    Loading subroutines from files on the fly

```

1.128 jade.guide/Function Call Forms

```

Function Call Forms
.....

```

The first element of a function call form is the name of the function, this can be either a symbol (in which case the symbol's function value is indirected through to get the real function definition) or a lambda expression (see

```

    Lambda Expressions
).

```

Any other elements of the list are forms to be evaluated (in left to right order) and their values become the arguments to the function. The function is applied to these arguments and the result that it returns becomes the value of the form.

For example, consider the form `'(/ 100 (1+ 4))'`. This is a function call to the function `'/'`. First the `'100'` form is evaluated: it returns the value `'100'`, next the form `'(1+ 4)'` is evaluated. This is also a function call and computes to a value of `'5'` which becomes the second argument to the `'/'` function. Now the `'/'` function is applied to its arguments of `'100'` and `'5'` and it returns the value `'20'` which then becomes the value of the form `'(/ 100 (1+ 4))'`.

```

(/ 100 (1+ 4))
== (/ 100 5)
=> 20

```

Or another example,

```

(+ (- 10 (1- 7)) (* (1+ 2) 4))
== (+ (- 10 6) (* (1+ 2) 4))
== (+ 4 (* (1+ 2) 4))
== (+ 4 (* 3 4))
== (+ 4 12)

```

=> 16

1.129 jade.guide/Macro Call Forms

Macro Call Forms

.....

Macros are source code expansions, the general idea is that a macro is a function which using the unevaluated arguments applied to it, computes another form (the expansion of the macro and its arguments) which is then evaluated to provide the value of the form. For more details see

Macros

.

1.130 jade.guide/Special Forms

Special Forms

.....

Special forms are built-in functions which the evaluator knows must be handled specially. The main difference between a special form and a function is that the arguments applied to a special form are **not** automatically evaluated -- if necessary the special form will evaluate arguments itself. This will be noted in the documentation of the special form.

Special forms are generally used to provide control structures, for example, all of the conditional constructs are special forms (if all of their arguments, including the forms to be conditionally evaluated, were evaluated automatically this would defeat the object of being conditional!).

The special forms supported by Jade are: `'and'`, `'catch'`, `'cond'`, `'defconst'`, `'defmacro'`, `'defun'`, `'defvar'`, `'error-protect'`, `'function'`, `'if'`, `'let'`, `'let*'`, `'or'`, `'progl'`, `'prog2'`, `'progn'`, `'quote'`, `'setq'`, `'setq-default'`, `'unless'`, `'unwind-protect'`, `'when'`, `'while'`, `'with-buffer'`, `'with-window'`.

1.131 jade.guide/Autoload Forms

Autoload Forms

.....

Not all modules of Jade are needed at once, autoload forms provide a

means of marking that a function (or macro) is contained by a specific file of Lisp code. The first time that the function is accessed the `autoload` form will be evaluated; this loads the file that the function is contained by then re-evaluates the list form.

By then the `autoload` form will have been overwritten in the symbol's function slot by the true function (when it was loaded) so the form will execute properly.

An `autoload` form is a list whose first element is the symbol `'autoload'`, for full details see
Autoloading
.

1.132 jade.guide/Self-Evaluating Forms

Self-Evaluating Forms

The computed value of any form which is not a symbol or a list will simply be the form itself and the form is said to be a "self-evaluating form".

Usually the only forms to be evaluated in this way will be numbers, strings and vectors (since they are the only other data types which have read syntaxes) but the effect is the same for other types of data.

This means that forms you know are self-evaluating do not have to be quoted to be used as constants (like lists and symbols do).

```
"foo"
=> "foo"

(eval (current-buffer))
=> #<buffer programmer.texi>
```

1.133 jade.guide/Quoting

Quoting

As the above sections explain some types of Lisp object have special meaning to the Lisp evaluator (namely the symbol and list types) this means that if you want to refer to a symbol or a list in a program you can't (yet) because the evaluator will treat the form as either a variable reference or a function call respectively.

To get around this Lisp uses something called "quoting", the `'quote'` special form simply returns its argument, without evaluating it. For

example,

```
(quote my-symbol)
=> my-symbol
```

the 'quote' form prevents the 'my-symbol' being treated as a variable -- it is effectively 'hidden' from the evaluator.

Writing 'quote' all the time would be a bit boring so there is a shortcut: the Lisp reader treats any form X preceded by a single quote character (``') as the form '(quote X)'. So the example above would normally be written as,

```
'my-symbol
=> my-symbol
```

- Special Form: quote FORM

This special form returns its single argument without evaluating it. This is used to "quote" constant objects to prevent them from being evaluated.

1.134 jade.guide/Control Structures

Control Structures

=====

Control structures are special forms or macros which control which forms get evaluated, when they get evaluated and the number of times to evaluate them. This includes conditional structures, loops, etc...

The simplest control structures are the sequencing structures; they are used to evaluate a list of forms in left to right order.

Sequencing Structures

Evaluating several forms in sequence

Conditional Structures

Making decisions based on truth values

Looping Structures

'while' loops

Non-Local Exits

Exiting from several levels of evaluation

1.135 jade.guide/Sequencing Structures

Sequencing Structures

Each of the special forms in this section simply evaluates its argument forms in left-to-right order. The only difference is the result they return.

The most widely used sequencing special form is `'progn'`: it evaluates all its argument forms and returns the computed value of the last one. Many other control structures are said to perform an "implicit progn", this means that they call `'progn'` with a list of forms.

`'progn'` in Lisp is nearly analogous to a `'begin...end'` block in Pascal; it is used in much the same places -- to allow you to evaluate a sequence of form where only one form was allowed (for example the true clause of an `'if'` structure).

- Special Form: `progn FORMS...`

All of the FORMS are evaluated sequentially (from left-to-right), the result of the last evaluated FORM is the return value of this structure. If no arguments are given to `'progn'` it returns `'nil'`.

```
(progn 'one (+ 1 1) "three")
=> "three"
```

```
(progn)
=> nil
```

- Special Form: `progl FIRST FORMS...`

This special form evaluates its FIRST form then performs an implicit progn on the rest of its arguments. The result of this structure is the computed value of the first form.

```
(progl 'one (+ 1 1) "three")
=> one
```

- Special Form: `prog2 FIRST SECOND FORMS...`

This is similar to `'progl'` except that the evaluated value of its SECOND form is returned.

The FIRST form is evaluated, then its SECOND, then it performs an implicit progn on the remaining arguments.

```
(prog2 'one (+ 1 1) "three")
=> 2
```

1.136 jade.guide/Conditional Structures

Conditional Structures

Lisp provides a number of conditional constructs, the most complex of

which (`'cond'`) will take a list of conditions, the first of which is `'t'` then has its associated list of forms evaluated. Theoretically this is the only conditional special form necessary -- the rest could be implemented as macros.

- Special Form: `if` CONDITION TRUE-FORM ELSE-FORMS...

The `'if'` construct is the nearest thing in Lisp to the "if-then-else" construct found in most programming languages.

First the CONDITION form is evaluated, if it returns `'t'` (not `'nil'`) the TRUE-FORM is evaluated and its result returned. Otherwise the result of an implicit progn on the ELSE-FORMS is returned. If there are no ELSE-FORMS `'nil'` is returned.

Note that one of the TRUE-FORM or the ELSE-FORMS is completely ignored -- it is not evaluated.

```
(if (special-form-p 'if)
    "'if' is a special form"
    "'if' is not a special form")
=> "'if' is a special form"
```

- Special Form: `when` CONDITION TRUE-FORMS...

CONDITION is evaluated, if it is `'t'` the result of an implicit progn on the TRUE-FORMS is returned, otherwise `'nil'` is returned.

```
(when t
  (message "Pointless")
  'foo)
=> foo
```

- Special Form: `unless` CONDITION ELSE-FORMS...

This special forms first evaluates CONDITION, if its computed value is not `'nil'` its value is returned. Otherwise the ELSE-FORMS are evaluated sequentially, the value of the last is returned.

- Special Form: `cond` CLAUSE...

The `'cond'` special form is used to choose between an arbitrary number of conditions. Each CLAUSE is a list; its car is the CONDITION the list which is the cdr of the CLAUSE is the BODY-FORMS. This means that each CLAUSE looks something like:

```
(CONDITION BODY-FORMS...)
```

and a whole `'cond'` form looks like:

```
(cond
  (CONDITION-1 BODY-FORMS-1...)
  (CONDITION-2 BODY-FORMS-2...)
  ...)
```

The CONDITION in each CLAUSE is evaluated in sequence (CONDITION-1, then CONDITION-2, ...), the first one which evaluates to a non-`'nil'` has an implicit progn performed on its BODY-FORMS, the value of which is the value returned by the `'cond'` form.

If the true CONDITION has no BODY-FORMS the value returned by 'cond' is the value of the CONDITION. If none of the clauses has a non-'nil' CONDITION the value of the 'cond' is 'nil'.

Often you want a "default" clause; one which has its BODY-FORMS to be evaluated if none of the other clauses are true. The way to do this is to add a clause with a CONDITION of 't' and BODY-FORMS of whatever you want the default action to be.

```
(cond
  ((stringp buffer-list))      ;Clause with no BODY-FORMS
  ((consp buffer-list)
   (setq x buffer-list)      ;Two BODY-FORMS
   t)
  (t                          ;Default clause
   (error "`buffer-list' is corrupted!")))
=> t
```

All of the other conditionals can be written in terms of 'cond',

```
(if C T E...) == (cond (C T) (t E...))

(when C T...) == (cond (C T...))

(unless C E...) == (cond (E) (t E...))
```

There are also a number of special forms which combine conditions together by the normal logical rules.

- Special Form: or FORMS...

The first of the FORMS is evaluated, if it is non-'nil' its value becomes the value of the 'or' form and no more of 'forms' are evaluated. Otherwise this step is repeated for the next member of FORMS.

If all of the FORMS have been evaluated and none have a non-'nil' value 'nil' becomes the value of the 'or' form.

If there are no FORMS 'nil' is returned.

```
(or nil 1 nil (beep))      ;`(beep)' won't be evaluated
=> 1
```

- Special Form: and FORMS...

The first of the FORMS is evaluated. If it is 'nil' no more of the FORMS are evaluated and 'nil' becomes the value of the 'and' structure. Otherwise the next member of FORMS is evaluated and its value tested. If none of the FORMS are 'nil' the computed value of the last member of FORMS becomes the value of the 'and' form.

```
(and 1 2 nil (beep))      ;`(beep)' won't be evaluated
=> nil

(and 1 2 3)                ;All forms are evaluated
=> 3
```

- Function: not OBJECT

This function inverts the boolean value of its argument. If OBJECT is non-`'nil'`, `'nil'` is returned, otherwise `'t'` is returned.

```
(not nil)
=> t

(not t)
=> nil

(not 42)
=> nil
```

1.137 jade.guide/Looping Structures

Looping Structures

Jade's version of Lisp has only one structure for looping -- a "while" loop similar to those found in most programming languages.

- Special Form: `while` CONDITION BODY-FORMS...

The CONDITION form is evaluated. If it is non-`'nil'` an implicit `progn` is performed on the BODY-FORMS and the whole thing is repeated again.

This continues until the CONDITION form evaluates to `'nil'`. The value of any `'while'` structure is `'nil'`.

`'while'` can be recursively defined in terms of `'when'`:

```
(while C B ...)
==
(when C (progn B ... (while C B ...)))

;; Step through a list X
(while X
  ;; Do something with the current element, '(car X)'
  (setq X (cdr X)))
```

1.138 jade.guide/Non-Local Exits

Non-Local Exits

A "non-local exit" is a transfer of control from the current point of evaluation to a different point (somewhat similar to the much-maligned `'goto'` statement in some imperative languages).

Non-local exits can either be used explicitly (`'catch'` and `'throw'`) or implicitly (`errors`).

Catch and Throw	Programmed non-local exits
Function Exits	Returning values from a function
Cleanup Forms	Forms which will always be evaluated
Errors	Signalling that an error occurred

1.139 jade.guide/Catch and Throw

Catch and Throw

.....

The 'catch' and 'throw' structures are used to perform explicit transfers of control. First a 'catch' form is used to setup a "tag", this acts like a label for the C language's 'goto' statement. To transfer control a 'throw' form is then used to transfer to the named tag. The tag is destroyed and the 'catch' form exits with the value provided by the 'throw'.

In a program this looks like,

```
(catch 'TAG
  ;; Forms which may 'throw' back to TAG
  ...
  (throw 'TAG VALUE)
  ;; Control has now passed to the 'catch',
  ;; no more forms in this progn will be evaluated.
  ...)
=> VALUE
```

where TAG is the tag to be used (this is normally a symbol) and VALUE is the result of the 'catch' form.

When a throw actually happens all catches in scope are searched for one with a tag which is 'eq' to the tag in the throw. If more than one exists the most-recent is chosen. Now that the catch has been located the environment is 'wound-back' to the catch's position (i.e. local variables are unbound, cleanup forms removed, unused catches forgotten, etc...) and all Lisp constructs between the current point of control and the catch are exited.

For example,

```
(let
  ((test 'outer))
```

```
(cons (catch 'foo
      (let
        ((test 'inner))
        (throw 'foo test)
        (setq test 'unreachable))) ;Never reached
      test))
=> (inner . outer)
```

when the throw executes the second binding of 'test' is unwound and the first binding comes back into effect. For more details on variable binding see

Local Variables

.

Note that catch tags are *dynamically* scoped, the thrower does not have to be within the same lexical scope (this means you can throw through functions).

- Special Form: catch TAG BODY-FORMS...

This special form defines a catch tag which will be accessible while the BODY-FORMS are being evaluated.

TAG is evaluated and recorded as the tag for this catch. Next the BODY-FORMS are evaluated as an implicit progn. The value of the 'catch' form is either the value of the progn, or, if a 'throw' happened, the value specified in the THROW form.

Before exiting the tag installed by this form is removed.

- Function: throw TAG &optional CATCH-VALUE

This function transfers the point of control to the catch form with a tag which is 'eq' to TAG. The value returned by this catch form is either CATCH-VALUE or 'nil' if CATCH-VALUE is undefined.

If there is no catch with a tag of TAG an error is signalled and the editor returns to the top-level of evaluation.

1.140 jade.guide/Function Exits

Function Exits

.....

It is often useful to be able to immediately return control from a function definition (like the C 'return' statement). Jade's version of Lisp has the 'return' function for this.

- Function: return &optional VALUE

This function transfers control out of the most-recent lambda-expression (i.e. a function or macro definition) so that the result of the lambda-expression is VALUE.

```
(funcall '(lambda () (return 'x) 'y))
=> x
```

The `'y'` form is never evaluated since control is passed straight from the `'(return 'y)'` form back to the `'funcall'` form.

1.141 jade.guide/Cleanup Forms

Cleanup Forms

.....

It is sometimes necessary to be sure that a certain form is **always** evaluated, even when a non-local exit would normally bypass that form. The `'unwind-protect'` special form is used to stop this happening.

- Special Form: `unwind-protect BODY-FORM CLEANUP-FORMS...`

The BODY-FORM is evaluated, if it exits normally the CLEANUP-FORMS are evaluated sequentially then the value which the BODY-FORM returned becomes the value of the `'unwind-protect'` form. If the BODY-FORM exits abnormally though (i.e. a non-local exit happened) the CLEANUP-FORMS are evaluated anyway and the non-local exit continues.

One use of this is to ensure that an opened file is always closed, for example,

```
(catch 'foo
  (unwind-protect
    (let
      ((temporary-file (open (tmp-file-name) "w")))
      ;; Use 'temporary-file'
      (write temporary-file "A test\n")
      ;; Now force a non-local exit
      (throw 'foo))
    ;; This is the CLEANUP-FORM it will *always*
    ;; be evaluated no matter what happens.
    (close temporary-file)))
=> nil
```

1.142 jade.guide/Errors

Errors

.....

Errors are a type of non-local exit; when a form can not be evaluated properly an error is normally "signalled". If an error-handler has been installed for that type of error control is unwound back to the handler and evaluation continues. If there is no suitable handler control is passed back to the event loop of the most-recent recursive edit and a suitable error message is printed.

- Function: `signal ERROR-SYMBOL DATA`

Signals that an error has happened. ERROR-SYMBOL is a symbol classifying the type of error, it should have a property 'error-message' (a string) which is the error message to be printed.

DATA is a list of objects which are relevant to the error -- they will be made available to any error-handler or printed with the error message otherwise.

```
(signal 'void-value ' (some-symbol))
  error--> Value as variable is void: some-symbol
```

- Variable: debug-on-error

This variable is consulted by the function 'signal'. If its value is either 't' or a list containing the ERROR-SYMBOL to 'signal' as one of its elements, the Lisp debugger is entered. When the debugger exits the error is signalled as normal.

When you expect an error to occur and need to be able to regain control afterwards the 'error-protect' form can be used.

- Special Form: error-protect BODY-FORM ERROR-HANDLERS...

'error-protect' evaluates the BODY-FORM with error handlers in place.

Each of the ERROR-HANDLERS is a list whose car is a symbol defining the type of error which this handler catches. The cdr of the list is a list of forms to be evaluated sequentially when the handler is invoked.

While the forms of the error handler are being evaluated the variable 'error-info' is bound to the value '(ERROR-SYMBOL . DATA)' (these were the arguments to the 'signal' form which caused the error).

The special value, the symbol 'error', in the car of one of the ERROR-HANDLERS will catch *all* types of errors.

```
(error-protect
  (signal 'file-error ' ("File not found" "/tmp/foo"))
  (file-error
   error-info)
  (error
   (setq x z)))          ;Default handler
  => (file-error "File not found" "/tmp/foo")
```

1.143 jade.guide/Variables

Variables

=====

In Lisp symbols are used to represent variables. Each symbol contains a slot which is used to contain the value of the symbol when it is used as a symbol.

The normal way to obtain the current value of a variable is simply to evaluate the symbol it lives in (i.e. write the name of the variable in your program).

- Function: `symbol-value VARIABLE`
This function returns the value of the symbol `VARIABLE` in the current environment.

Local Variables	Creating temporary variables
Setting Variables	Altering a variable's value
Scope and Extent	Technical jargon
Buffer-Local Variables	Variables with distinct values in each buffer.
Void Variables	Some variables have no values
Constant Variables	Variables which may not be altered
Defining Variables	How to define a variable before using it

1.144 jade.guide/Local Variables

Local Variables

A "local variable" is a variable which has a temporary value while a program is executing, for example, when a function is called the variables which are the names of its arguments are temporarily bound (a "binding" is a particular instance of a local variable) to the values of the arguments passed to the function. When the function call exits its arguments are unbound and the previous definitions of the variables come back into view.

Even if a variable has more than one binding still 'active' only the most recent is visible -- there is absolutely no way the previous bindings can be accessed until the bindings are unbound one-by-one.

A nice way of visualising variable binding is to think of each variable as a stack. When the variable is bound to, a new value is

pushed onto the stack, when it is unbound the top of the stack is popped. Similarly when the stack is empty the value of the variable is void (see

Void Variables

). Assigning a value to the variable (see

Setting Variables

) overwrites the top value on the stack with a new value. When the value of the variable is required it is simply read from the top of the stack.

Apart from function calls there are two special forms which perform variable binding (i.e. creating local variables), 'let' and 'let*'.

- Special Form: let BINDINGS BODY-FORMS...

'let' creates new variable bindings as specified by the BINDINGS argument then evaluates the BODY-FORMS in order. The variables are then unbound to their state before this 'let' form and the value of the implicit progn of the BODY-FORMS becomes the value of the 'let' form.

The BINDINGS argument is a list of the bindings to perform. Each binding is either a symbol, in which case that variable is bound to nil, or a list whose car is a symbol. The cdr of this list is a list of forms which, when evaluated, give the value to bind the variable to.

```
(setq foo 42)
=> 42
(let
  ((foo (+ 1 2))
   bar)
  ;; Body forms
  (setq foo (1+ foo)) ;This sets the new binding
  (cons foo bar))
=> (4 . nil)
foo
=> 42 ;The original values is back
```

Note that no variables are bound until all the new values have been computed (unlike in 'let*'). For example,

```
(setq foo 42)
=> 42
(let
  ((foo 100)
   (bar foo))
  (cons foo bar))
=> (100 . 42)
```

Although 'foo' is given a new binding this is not actually done until all the new bindings have been computed, hence 'bar' is bound to the *old* value of 'foo'.

- Special Form: let* BINDINGS BODY-FORMS...

This special form is exactly the same as 'let' except for one important difference: the new bindings are installed *as they are

computed*.

You can see the difference by comparing the following example with the last example in the 'let' documentation (above),

```
(setq foo 42)
=> 42
(let*
  ((foo 100)
   (bar foo))
  (cons foo bar))
=> (100 . 100) ;Using 'let*' this time
```

By the time the binding of 'bar' is computed the new binding of 'foo' has already been installed.

1.145 jade.guide/Setting Variables

Setting Variables

"Setting" a variable means to overwrite its current value (that is, the value of its most recent binding) with a new one. The old value is irretrievably lost (unlike when a new value is bound to a variable, see

Local Variables
)

- Special Form: `setq VARIABLE FORM ...`

The special form 'setq' is the usual method of altering the value of a variable. Each VARIABLE is set to the result of evaluating its corresponding FORM. The last value assigned becomes the value of the 'setq' form.

```
(setq x 20 y (+ 2 3))
=> 5
```

In the above example the variable 'x' is set to '20' and 'y' is set to the value of the form '(+ 2 3)' (5).

When the variable is marked as being buffer-local (see

Buffer-Local Variables
) the current buffer's instance of the variable is set.

- Function: `set VARIABLE NEW-VALUE`

The value of the variable VARIABLE (a symbol) is set to NEW-VALUE and the NEW-VALUE is returned.

This function is used when the VARIABLE is unknown until run-time, and therefore has to be computed from a form.

```
(set 'foo 20)
==
(setq foo 20)      ; `setq' means `set-quoted'
=> 20
```

1.146 jade.guide/Scope and Extent

Scope and Extent

In Jade's version of Lisp all variables have "indefinite scope" and "dynamic extent". What this means is that references to variables may occur anywhere in a program (i.e. bindings established in one function are not only accessible within that function, that's lexical scope) and that references may occur at any point in the time between the binding being created and it being unbound.

The combination of indefinite scope and dynamic extent is often termed "dynamic scope".

As an aside, Lisp objects have "indefinite extent", meaning that the object will exist for as long as there is a possibility of it being referenced (and possibly longer -- until the garbage collector runs).

Note that in Common Lisp only those variables declared 'special' have indefinite scope and dynamic extent.

Try not to abuse the dynamic scoping, although it is often very useful to be able to bind a variable in one function and use it in another this can be confusing if not controlled and documented properly.

A quick example of the use of dynamic scope,

```
(defun foo (x)
  (let
    ((foo-var (* x 20)))
    (bar x)
    ...

  (defun bar (y)
    ;; Since this function is called from
    ;; the function 'foo' it can refer
    ;; to any bindings which 'foo' can.
    (setq y (+ y foo-var))
    ...
```

1.147 jade.guide/Buffer-Local Variables

Buffer-Local Variables

It is often very useful to be able to give variables different values for different editor buffers -- most major modes need to record some buffer-specific information. Jade allows you to do this by giving a variable buffer-local bindings.

There are two strengths of buffer-local variables: you can either give a variable a buffer-local value in a single buffer, with other buffers treating the variable as normal, or a variable can be marked as being **automatically** buffer-local, each time the variable is set the current buffer's value of the variable is updated.

Each buffer maintains an alist of the symbols which have buffer-local values in the buffer and the actual values themselves, this alist may be read with the `'buffer-variables'` function.

When the value of a variable is referenced (via the `'symbol-value'` function) the current buffer's alist of local values is examined for a binding of the variable being referenced; if one is found that is the value of the variable, otherwise the "default value" (the value stored in the symbol's value cell) is used.

Setting a variable also searches for a buffer-local binding; if one exists its value is modified, not the default value. If the variable has previously been marked as being automatically buffer-local (by `'make-variable-buffer-local'`) a buffer-local binding is automatically created if one doesn't already exist.

Currently there is one main problem with buffer-local variables: they can't have temporary values bound to them (or rather, they can but I guarantee it won't work how you expect), so for the time being, don't try to bind local values (with `'let'` or `'let*'`) to a buffer-local variable.

- Function: `make-local-variable` SYMBOL

This function gives the variable SYMBOL a buffer-local binding in the current buffer. The value of this binding will be the same as the variable's default value.

If SYMBOL already has a buffer-local value in this buffer nothing happens.

Returns SYMBOL.

- Function: `make-variable-buffer-local` SYMBOL

This function marks the variable SYMBOL as being automatically buffer-local.

This means that any attempts at setting the value of SYMBOL will actually set the current buffer's local value (if necessary a new buffer-local binding will be created in the buffer).

Returns SYMBOL.

```
(make-variable-buffer-local 'buffer-modtime)
=> buffer-modtime
```

- Function: `default-value SYMBOL`

This function returns the default value of the variable `SYMBOL`.

```
(setq foo 'default)
=> default
(make-local-variable 'foo)      ;Create a value in this buffer
=> foo
(setq foo 'local)
=> local
foo
=> local
(symbol-value 'foo)
=> local
(default-value 'foo)
=> default
```

- Function: `default-boundp SYMBOL`

Returns `'t`' if the variable `SYMBOL` has a non-void default value.

- Special Form: `setq-default SYMBOL FORM ...`

Similar to the `'setq'` special form except that the default value of each `VARIABLE` is set. In non-buffer-local symbols there is no difference between `'setq'` and `'setq-default'`.

- Function: `set-default SYMBOL NEW-VALUE`

Sets the default value of the variable `SYMBOL` to `NEW-VALUE`, then returns `NEW-VALUE`.

- Function: `kill-local-variable SYMBOL`

This function removes the buffer-local binding of the variable `SYMBOL` from the current buffer (if one exists) then returns `SYMBOL`.

- Function: `kill-all-local-variables`

This function removes all the buffer-local bindings associated with the current buffer. Subsequently, any buffer-local variables referenced while this buffer is current will use their default values.

The usual way to define an automatically buffer-local variable is to use `'defvar'` and `'make-variable-buffer-local'`, for example,

```
(defvar my-local-variable DEFAULT-VALUE
  "Doc string for 'my-local-variable'.")
(make-variable-buffer-local 'my-local-variable)
```

Note that if you want to reference the value of a buffer-local variable in a buffer other than the current buffer, use the `'with-buffer'` special form (see

The Current Buffer
). For example, the

form,

```
(with-buffer OTHER-BUFFER SOME-VARIABLE)
```

will produce the value of the variable SOME-VARIABLE in the buffer OTHER-BUFFER.

1.148 jade.guide/Void Variables

Void Variables

A variable which has no value is said to be "void", attempting to reference the value of such a symbol will result in an error. It is possible for the most recent binding of a variable to be void even though the inactive bindings may have values.

- Function: `boundp VARIABLE`
Returns `'t'` if the symbol `VARIABLE` has a value, `'nil'` if its value is void.
- Function: `makunbound VARIABLE`
This function makes the current binding of the symbol `VARIABLE` be void, then returns `VARIABLE`.

```
(setq foo 42)
=> 42
foo
=> 42
(boundp 'foo)
=> t
(makunbound 'foo)
=> foo
(boundp 'foo)
=> nil
foo
error--> Value as variable is void: foo
```

1.149 jade.guide/Constant Variables

Constant Variables

In Lisp constants are represented by variables which have been marked as being read-only. Any attempt to alter the value of a constant results in an error.

Two of the most commonly used constants are `'nil'` and `'t'`.

- Function: `set-const-variable VARIABLE &optional READ-WRITE`
This function defines whether or not the value of the symbol `VARIABLE` may be modified. If `READ-WRITE` is `'nil'` or undefined the variable is marked to be constant, otherwise it's marked to be a

normal variable. The value returned is VARIABLE.

- Function: `const-variable-p` VARIABLE
Returns `t` if the value of the symbol VARIABLE may be altered, `nil` otherwise.

Constants may behave a bit strangely when you compile the program they are used in: the value of the constant is likely to be hardwired into the compiled functions it is used in, and the constant is unlikely to be `eq` to itself!

The compiler assumes that constant is always the same, whenever it is evaluated. It may even be evaluated more than once. See

Compiled Lisp

.

The special form `defconst` can be used to define constants, see

Defining Variables

.

1.150 jade.guide/Defining Variables

Defining Variables

The special forms `defvar` and `defconst` allow you to define the global variables which will be used in a program. This is entirely optional; it is highly recommended though.

- Special Form: `defvar` VARIABLE FORM [DOC-STRING]
This special form defines a global variable, the symbol VARIABLE. If the value of VARIABLE is void the FORM is evaluated and its value is stored as the value of VARIABLE (note that the default value is modified, never a buffer-local value).

If the DOC-STRING argument is defined it is a string documenting VARIABLE. This string is then stored as the symbol's `'variable-documentation'` property and can be accessed by the `'describe-variable'` function.

```
(defvar my-variable '(x y)
  "This variable is an example showing the usage of the 'defvar'
  special form.")
=> my-variable
```

- Special Form: `defconst` CONSTANT FORM [DOC-STRING]
`'defconst'` defines a constant, the symbol CONSTANT. Its value (in the case of a buffer-local symbol, its default value) is set to the result of evaluating FORM. Note that unlike `'defvar'` the value of the symbol is **always** set, even if it already has a value.

The DOC-STRING argument, if defined, is the documentation string

for the constant.

```
(defconst the-answer 42
  "An example constant.")
=> the-answer
```

See

Constant Variables

.

1.151 jade.guide/Functions

Functions

=====

A "function" is a Lisp object which, when applied to a sequence of argument values, produces a value -- the function's "result". It may also produce side-effects. All Lisp functions return results -- there is nothing like a procedure in Pascal.

Functions are the main building-block in Lisp programs, each program is usually a system of inter-related functions.

There are two types of function: "primitive functions" are functions written in the C language, these are sometimes called built-in functions, the object containing the C code itself is called a "subr". All other functions are written in Lisp.

- Function: functionp OBJECT
Returns 't' if OBJECT is a function (i.e. it can be used as the function argument of 'funcall').

```
(functionp 'set)
=> t
```

```
(functionp 'setq)
=> nil
```

```
(functionp #'(lambda (x) (+ x 2)))
=> t
```

Lambda Expressions

Structure of a function object

Named Functions

Functions can be named by symbols,

Anonymous Functions

Or they can be un-named

Predicate Functions

Functions which return boolean values

Defining Functions

How to write a function definition

Calling Functions

Functions can be called by hand

Mapping Functions

Map a function to the elements of a list

1.152 jade.guide/Lambda Expressions

Lambda Expressions

"Lambda expressions" are used to create an object of type function from other Lisp objects, it is a list whose first element is the symbol 'lambda'. All functions written in Lisp (as opposed to the primitive functions in C) are represented by a lambda expression.

Note that a lambda expression is *not* an expression, evaluating a lambda expression will give an error (unless there is a function called 'lambda').

The format of a lambda expression is:

```
(lambda LAMBDA-LIST [DOC] [INTERACTIVE-DECLARATION] BODY-FORMS... )
```

Where LAMBDA-LIST is the argument specification of the function, DOC is an optional documentation string, INTERACTIVE-DECLARATION is only required by editor commands (see

Commands

) and the BODY-FORMS is the

actual function code (when the function is called each form is evaluated in sequence, the last form's value is the result returned by the function).

The LAMBDA-LIST is a list, it defines how the argument values applied to the function are bound to local variables which represent the arguments within the function. At its simplest it is simply a list of symbols, each symbol will have the corresponding argument value bound to it. For example, the lambda list,

```
(lambda (x y) (+ x y))
```

takes two arguments, 'x' and 'y'. When this function is called with two arguments the first will be bound to 'x' and the second to 'y' (then the function will return their sum).

To complicate matters there are several "lambda-list keywords" which modify the meaning of symbols in the lambda-list. Each keyword is a symbol whose name begins with an ampersand, they are:

'&optional'

All the variables following this keyword are considered "optional" (all variables before the first keyword are "required": an error will be signalled if a required argument is undefined in a function call). If an optional argument is undefined it will simply be given the value 'nil'.

Note that optional arguments must be specified if a later optional argument is also specified. Use 'nil' to explicitly show that an optional argument is undefined.

For example, if a function 'foo' takes two optional arguments and you want to call it with only the second argument defined, the first argument must be specified as 'nil' to ensure that the correct argument value is bound to the correct variable.

```
(defun foo (&optional arg-1 arg-2)
  ...

  (foo nil arg-2-value) ;Leave the first argument undefined
```

'&rest'

The '&rest' keyword allows a variable number of arguments to be applied to a function, all the argument values which have not been bound to argument variables are simply consed into a list and bound to the variable after the '&rest' keyword. For example, in,

```
(lambda (x &rest y) ...)
```

the first argument, 'x', is required. Any other arguments applied to this function are made into a list and this list is bound to the 'y' variable.

When a function represented by a lambda-list is called the first thing that happens is to bind the argument values to the argument variables. The LAMBDA-LIST and the list of argument values applied to the function are worked through in parallel. Any required arguments which are left undefined when the end of the argument values has been reached causes an error.

After the arguments have been processed the BODY-FORMS are evaluated by an implicit progn, the value of which becomes the value of the function call. Finally, all argument variables are unbound and control passes back to the caller.

1.153 jade.guide/Named Functions

Named Functions

Functions are normally associated with symbols, the name of the symbol being the same as the name of its associated function. Each

symbol has a special function cell (this is totally separate from the symbol's value as a variable -- variables and functions may have the same name without any problems occurring) which is used to store the function's definition, either a lambda expression (see

```

Lambda Expressions
) or a subr (C code) object.

```

The evaluator knows to indirect through the function value of a symbol in any function call (see

```

Function Call Forms
) so the normal way

```

to call a function is simply write its name as the first element in a list, any arguments making up the other elements in the list. See

```

List Forms
.

```

The functions and special forms which take functions as their arguments (i.e. 'funcall') can also take symbols. For example,

```

(funcall 'message "An example")
==
(message "An example")

```

- Function: symbol-function SYMBOL
Returns the value of the function cell in the symbol SYMBOL.

```

(symbol-function 'symbol-function)
=> #<subr symbol-function>

```

- Function: fboundp SYMBOL
This function returns 't' if the symbol SYMBOL has a non-void value in its function cell, 'nil' otherwise.

```

(fboundp 'setq)
=> t

```

- Function: fset SYMBOL NEW-VALUE
Sets the value of the function cell in the symbol SYMBOL to NEW-VALUE, then returns NEW-VALUE.

```

This function is rarely used, see
Defining Functions
.

```

- Function: fmakunbound SYMBOL
This function makes the value of the function cell in SYMBOL void, then returns SYMBOL.

1.154 jade.guide/Anonymous Functions

Anonymous Functions

When giving function names as arguments to functions it is useful to give an actual function **definition** (i.e. a lambda expression) instead of the name of a function.

In Lisp, unlike most other programming languages, functions have no inherent name. As seen in the last section named-functions are created by storing a function in a special slot of a symbol, if you want, a function can have many different names: simply store the function in many different symbols!

So, when you want to pass a function as an argument there is the option of just writing down its definition. This is especially useful with functions like 'mapcar' and 'delete-if'. For example, the following form removes all elements from the LIST which are even and greater than 20.

```
(setq LIST (delete-if #'(lambda (x)
                        (and (zerop (% x 2))
                             (> x 20)))
                    LIST))
```

The lambda expression is very simple, it combines two predicates applied to its argument.

Note that the function definition is quoted by '#'', not the normal '''. This is a special shortcut for the 'function' special form (like '' is a shortcut to 'quote'). In general, '#X' is expanded by the Lisp reader to '(function X)'.

- Special Form: function ARG

This special form is nearly identical to the 'quote' form, it always returns its argument without evaluating it. The difference is that the Lisp compiler knows to compile the ARG into a byte-code form (unless ARG is a symbol in which case it is not compiled).

What this means is when you have to quote a function, use the '#'' syntax.

1.155 jade.guide/Predicate Functions

Predicate Functions

In Lisp, a function which returns a boolean 'true' or boolean 'false' value is called a "predicate". As is the convention in Lisp a value of 'nil' means false, anything else means true. The symbol 't' is often used to represent a true value (in fact, sometimes the symbol 't' should be read as **any* non-'nil' value*).

Another Lisp convention is that the names of predicate functions should be the concept the predicate is testing for and either 'p' or

`-p'.

The `p' variant is used when the concept name does not contain any hyphens.

For example a predicate to test for the concept "const-variable" (a variable which has a constant value, see

Constant Variables

) would be

called `const-variable-p'. On the other hand a predicate to test for the concept "buffer" (a Lisp object which is a buffer) would be called `bufferp'.

1.156 jade.guide/Defining Functions

Defining Functions

Named functions are normally defined by the `defun' special form.

- Special Form: defun NAME LAMBDA-LIST BODY-FORMS...

`defun' initialises the function definition of the symbol NAME to the lambda expression resulting from the concatenation of the symbol `lambda', LAMBDA-LIST and the BODY-FORMS. So,

```
(defun foo (x y)
  ...
  ==
  (fset 'foo #'(lambda (x y)
    ...
```

The BODY-FORMS may contain a documentation string for the function as its first form and an interactive calling specification as its first (if there is no doc-string) or second form if the function may be called interactively by the user (see

Commands

).

An example function definition (actually a command) taken from Jade's source is,

```
(defun upcase-word (count)
  "Makes the next COUNT words from the cursor upper-case."
  (interactive "p")
  (let
    ((pos (forward-word count))
     (upcase-area (cursor-pos) pos)
     (goto-char pos)))
```

1.157 jade.guide/Calling Functions

Calling Functions

Most of the time function calls are done by the evaluator when it detects a function call form (see

List Forms

); when the function to be

called is not known until run-time it is easier to use a special function to call the function directly than create a custom form to apply to the 'eval' function.

- Function: `funcall FUNCTION &rest ARGS`
Applies the argument values `ARGS` to the function `FUNCTION`, then returns its result.

Note that the argument values `ARGS` are **not** evaluated again. This also means that 'funcall' can **not** be used to call macros or special forms -- they would need the unevaluated versions of `ARGS`, which are not available to 'funcall'.

```
(funcall '+ 1 2 3)
=> 6
```

- Function: `apply FUNCTION &rest ARGS`
Similar to 'funcall' except that the last of its arguments is a **list** of arguments which are appended to the other members of `ARGS` to form the list of argument values to apply to the function `FUNCTION`.

Constructs a list of arguments to apply to the function `FUNCTION` from `ARGS`.

1.158 jade.guide/Mapping Functions

Mapping Functions

A "mapping function" applies a function to each of a collection of objects. Jade currently has two mapping functions, 'mapcar' and 'mapc'.

- Function: `mapcar FUNCTION LIST`
Each element in the list `LIST` is individually applied to the function `FUNCTION`. The values returned are made into a new list which is returned.

The `FUNCTION` should be able to be called with one argument.

```
(mapcar '1+ '(1 2 3 4 5))
=> (2 3 4 5 6)
```

- Function: `mapc` FUNCTION LIST

Similar to `'mapcar'` except that the values returned when each element is applied to the function FUNCTION are discarded. The value returned is LIST.

This function is generally used where the side effects of calling the function are the important thing, not the results.

The two following functions are also mapping functions of a sort. They are variants of the `'delete'` function (see `Modifying Lists`) and

use predicate functions to classify the elements of the list which are to be deleted.

- Function: `delete-if` PREDICATE LIST

This function is a variant of the `'delete'` function. Instead of comparing each element of LIST with a specified object, each element of LIST is applied to the predicate function PREDICATE. If it returns `'t'` (i.e. not `'nil'`) then the element is destructively removed from LIST.

```
(delete-if 'stringp '(1 "foo" 2 "bar" 3 "baz"))
=> (1 2 3)
```

- Function: `delete-if-not` PREDICATE LIST

This function does the inverse of `'delete-if'`. It applies PREDICATE to each element of LIST, if it returns `'nil'` then the element is destructively removed from the list.

```
(delete-if-not 'stringp '(1 "foo" 2 "bar" 3 "baz"))
=> ("foo" "bar" "baz")
```

1.159 jade.guide/Macros

Macros

=====

"Macros" are used to extend the Lisp language, they are basically a function which instead of returning its value, return a new form which will produce the macro call's value when evaluated.

When a function being compiled calls a macro the macro is expanded immediately and the resultant form is open-coded into the compiler's output.

Defining Macros

Macros are defined like functions

Macro Expansion

How macros are used by the evaluator

Compiling Macros

The compiler expands macros at compile-time.

1.160 jade.guide/Defining Macros

Defining Macros

Macros are defined in the same style as functions, the only difference is the name of the special form used to define them.

A macro object is a list whose car is the symbol 'macro', its cdr is the function which creates the expansion of the macro when applied to the macro calls unevaluated arguments.

- Special Form: defmacro NAME LAMBDA-LIST BODY-FORMS...

Defines the macro stored in the function cell of the symbol NAME. LAMBDA-LIST is the lambda-list specifying the arguments to the macro (see

Lambda Expressions

) and BODY-FORMS are the forms

evaluated when the macro is expanded. The first of BODY-FORMS may be a documentation string describing the macro's use.

Here is a simple macro definition, it is a possible definition for the 'when' construct (which might even be useful if 'when' wasn't already defined as a special form...),

```
(defmacro when (condition &rest body)
  "Evaluates CONDITION, if it's non-'nil' evaluates the BODY forms."
  (list 'if condition (cons 'progn body)))
```

When a form of the type '(when C B ...)' is evaluated the macro definition of 'when' expands to the form '(if C (progn B ...))' which is then evaluated to perform my when-construct.

When you define a macro ensure that the forms which produce the expansion have no side effects; it would fail spectacularly when you attempt to compile your program!

1.161 jade.guide/Macro Expansion

Macro Expansion

When a macro call is detected (see
 List Forms
) the function which is
 the cdr of the macro's definition (see
 Defining Macros
) is applied to
 the macro call's arguments. Unlike in a function call, the arguments
 are **not evaluated**, the actual forms are the arguments to the macro's
 expansion function. This is so these forms can be rearranged by the
 macro's expansion function to create the new form which will be
 evaluated.

There is a function which performs macro expansion, its main use is
 to let the Lisp compiler expand macro calls at compile time.

- Function: macroexpand FORM &optional ENVIRONMENT

If FORM is a macro call 'macroexpand' will expand that call by
 calling the macro's expansion function (the cdr of the macro
 definition). If this expansion is another macro call the process
 is repeated until an expansion is obtained which is not a macro
 call, this form is then returned.

The optional ENVIRONMENT argument is an alist of macro definitions
 to use as well as the existing macros; this is mainly used for
 compiling purposes.

```
(defmacro when (condition &rest body)
  "Evaluates CONDITION, if it's non-'nil' evaluates the BODY
  forms."
  (list 'if condition (cons 'progn body)))
=> when

(macroexpand '(when x (setq foo bar)))
=> (if x (progn (setq foo bar)))
```

1.162 jade.guide/Compiling Macros

Compiling Macros

Although it may seem odd that macros return a form to produce a
 result and not simply the result this is their most important feature.
 It allows the expansion and the evaluation of the expansion to happen
 at different times.

The Lisp compiler makes use of this; when it comes across a macro
 call in a form it is compiling it uses the 'macroexpand' function to
 produce the expansion of that form which it then compiles straight into
 the object code. Obviously this is good for performance (why evaluate
 the expansion every time it is needed when once will do?).

Some rules do need to be observed to make this work properly:

- * When the compiler compiles a file it remembers the macros which have been defined by that file; it can only expand a macro call if the definition of the macro appears before the macro call itself (it can't read your mind).
- * The macro expansion function (i.e. the definition of the macro) should not have any side effects or evaluate its arguments (the value of a symbol at compile-time probably won't be the same as its value at run-time).
- * Macros which are defined by another file must be loaded so they can be recognised. Use the 'require' function, the compiler will evaluate any top-level 'require' forms it sees to bring in any macro definitions used.

1.163 jade.guide/Streams

Streams

=====

A "stream" is a Lisp object which is either a data sink (an "output stream") or a data source (an "input stream"). In Jade all streams produce or consume sequences of 8-bit characters.

Streams are very flexible, functions using streams for their input and output do not need to know what type of stream it is. For example the Lisp reader (the 'read' function) takes an input stream as its one argument, it then reads characters from this stream until it has parsed a whole object. This stream could be a file, a position in a buffer, a function or even a string; the 'read' function can not tell the difference.

- Function: streamp OBJECT

This function returns 't' if its argument is a stream.

Input Streams

Types of input stream

Output Streams

Types of output stream

Input Functions

Functions to read from streams

Output Functions

How to output to a stream

1.164 jade.guide/Input Streams

Input Streams

These are the possible types of input stream, for the functions which use them see

Input Functions

.

'FILE'

Characters are read from the file object FILE, for the functions which manipulate file objects see

Files

.

'MARK'

The marker MARK points to the next character that will be read. Each time a character is read the position that MARK points to will be advanced to the following character. See

Marks

.

'BUFFER'

Reads from the position of the cursor in the buffer BUFFER. This position is advanced as characters are read.

'(BUFFER . POSITION)'

Characters are read from the position POSITION in the buffer BUFFER. POSITION is advanced to the next character as each character is read.

'FUNCTION'

Each time an input character is required the FUNCTION is called with no arguments. It should return the character read (an integer) or 'nil' if for some reason no character is available.

FUNCTION should also be able to 'unread' one character. When this happens the function will be called with one argument -- the value of the last character read. The function should arrange it so that the next time it is called it returns this character. A possible implementation could be,

```
(defvar ms-unread-char nil
  "If non-nil the character which was pushed back.")

(defun my-stream (&optional unread-char)
  (if unread-char
      (setq ms-unread-char unread-char)
      (if ms-unread-char
          (progl
            ms-unread-char
            (setq ms-unread-char nil))
          ;; Normal case -- read and return a character from somewhere
          ...
```

`'nil'`

Read from the stream stored in the variable `'standard-input'`.

It is also possible to use a string as an input stream. The string to be read from must be applied to the `'make-string-input-stream'` function and the result from this function used as the input stream.

- Function: `make-string-input-stream` STRING &optional START
Returns an input stream which will supply the characters of the string STRING in order starting with the character at position START (or from position zero if this argument is undefined).

```
(read (make-string-input-stream "(1 . 2)"))
=> (1 . 2)
```

- Variable: `standard-input`
The input stream which is used when no other is specified or is `'nil'`.

1.165 jade.guide/Output Streams

Output Streams

These are the different types of output stream, for the functions which use them see

Output Functions

.

`'FILE'`

Writes to the file object FILE. See
Files

.

`'MARK'`

Writes to the position pointed to by the marked MARK, then advances the position of the mark.

`'BUFFER'`

Writes to BUFFER at the position of the cursor in that buffer, which is then advanced.

`'(BUFFER . POSITION)'`

POSITION in the buffer BUFFER. POSITION is then moved over the written text.

`'(BUFFER . t)'`

Writes to the end of the buffer BUFFER.

`'FUNCTION'`

The function FUNCTION is called with one argument, either a string or a character. This should be used as the circumstances dictate. If the function returns a number it is the number of characters

actually used, otherwise it is assumed that all the characters were successful.

`'PROCESS'`

Writes to the standard input of the process object `PROCESS`. If `PROCESS` isn't running an error is signalled. See `Processes`.

`'t'`

Appends the character(s) to the end of the status line message.

`'nil'`

Write to the stream stored in the variable `'standard-output'`.

It is also possible to store the characters sent to an output stream in a string.

- Function: `make-string-output-stream`
Returns an output stream. It accumulates the text sent to it for the benefit of the `'get-output-stream-string'` function.
 - Function: `get-output-stream-string` `STRING-OUTPUT-STREAM`
Returns a string consisting of the text sent to the `STRING-OUTPUT-STREAM` since the last call to `GET-OUTPUT-STREAM-STRING` (or since this stream was created by `'make-string-output-stream'`).
- ```
(setq stream (make-string-output-stream))
=> (" " . 0)
(prinl keymap-path stream)
=> "(lisp-mode-keymap global-keymap)" . 64)
(get-output-stream-string stream)
=> "(lisp-mode-keymap global-keymap)"
```
- Variable: `standard-output`  
This variable contains the output stream which is used when no other is specified (or when the given output stream is `'nil'`).

## 1.166 jade.guide/Input Functions

### Input Functions

---

- Function: `read-char` `STREAM`  
Read and return the next character from the input stream `STREAM`. If the end of the stream is reached `'nil'` is returned.
  - Function: `read-line` `STREAM`  
This function reads one line of characters from the input stream `STREAM`, creates a string containing the line (including the newline character which terminates the line) and returns it.
-

If the end of stream is reached before any characters can be read 'nil' is returned, if the end of stream is reached but some characters have been read (but not the newline) these characters are made into a string and returned.

Note that unlike the Common Lisp function of the same name, the newline character is not removed from the returned string.

- Function: read STREAM

This function is the function which contains the Lisp reader (see

The Lisp Reader

). It reads as many characters from the input

stream STREAM as it needs to make the read syntax of a single Lisp object (see

Read Syntax

), this object is then returned.

- Function: read-from-string STRING &optional START

Reads one Lisp object from the string STRING, the first character is read from position START (or position zero).

```
(read-from-string STRING START)
```

```
==
```

```
(read (make-string-input-stream STRING START))
```

## 1.167 jade.guide/Output Functions

### Output Functions

- Function: write STREAM DATA &optional LENGTH

Writes the specified character(s) to the output stream STREAM.

DATA is either the character or the string to be written. If DATA is a string the optional argument LENGTH may specify how many characters are to be written. The value returned is the number of characters successfully written.

```
(write standard-output "Testing 1.. 2.. 3..")
```

```
-| Testing 1.. 2.. 3..
```

```
=> 19
```

- Function: copy-stream INPUT-STREAM OUTPUT-STREAM

This function copies all characters which may be read from

INPUT-STREAM to OUTPUT-STREAM. The copying process is not stopped until the end of the input stream is read. Returns the number of characters copied.

Be warned, if you don't choose the streams carefully you may get a deadlock which only an interrupt signal can break!

- Function: print OBJECT &optional STREAM

Outputs a newline character to the output stream STREAM, then



writes a textual representation of OBJECT to the stream.

If possible, this representation will be such that 'read' can turn it into an object structurally similar to OBJECT. This will \*not\* be possible if OBJECT does not have a read syntax.

OBJECT is returned.

```
(print '(1 2 3))
-|
-| (1 2 3)
=> (1 2 3)
```

- Function: `prinl OBJECT &optional STREAM`  
Similar to 'print' but no initial newline is output.

```
(prinl '(1 2 3))
-| (1 2 3)
=> (1 2 3)
```

```
(prinl '|(xy((z)|)) ;A strange symbol
-| \ (xy\ (\ (z\]
=> \ (xy\ (\ (z\]
```

- Function: `prinl-to-string OBJECT`  
Returns a string containing the characters that 'prinl' would output when it prints OBJECT.

```
(prinl-to-string '(1 2 3))
=> "(1 2 3)"
```

- Function: `princ OBJECT &optional STREAM`  
Prints a textual representation of OBJECT to the output stream STREAM. No steps are taken to create output that 'read' can parse and no quote characters surround strings.

```
(princ "foo")
-| foo
=> "foo"
```

```
(princ '|(xy((z)|))
-| (xy((z]
=> \ (xy\ (\ (z\]
```

- Function: `format STREAM TEMPLATE &rest VALUES`  
Writes to a stream, STREAM, a string constructed from the format string, TEMPLATE, and the argument VALUES.

If STREAM is 'nil' the resulting string will be returned, not written to a stream.

TEMPLATE is a string which may contain format specifiers, these are a '%' character followed by another character telling how to print the next of the VALUES. The following options are available

's'

Write the printed representation of the value without quoting

---

(as if from the `'princ'` function).

`'s'`

Write the printed representation *\*with\** quoting enabled (like the `'prin1'` function).

`'d'`

Output the value as a decimal number.

`'o'`

Write the value in octal.

`'x'`

In hexadecimal.

`'c'`

Write the character specified by the value.

`'%'`

Print a literal percent character. None of the VALUES are used.

The function works through the TEMPLATE a character at a time. If the character is a format specifier (a `'%'`) it inserts the correct string (as defined above) into the output. Otherwise, the character is simply put into the output stream.

If STREAM isn't `'nil'` (i.e. the formatted string is returned) the value of STREAM is returned.

```
(format nil "foo %S bar 0x%x" '(x . y) 255)
=> "foo (x . y) bar 0xff"
```

```
(format standard-output "The %s is %s!" "dog" "purple")
-| The dog is purple!
=> #<buffer *jade*>
```

## 1.168 jade.guide/Loading

Loading

=====

In Lisp, programs (also called "modules") are stored in files. Each file is a sequence of Lisp forms (known as "top-level forms"). Most of the top-level forms in a program will be definitions (i.e. function, macro or variable definitions) since generally each module is a system of related functions and variables.

Before the program can be used it has to be "loaded" into the editor's workspace; this involves reading and evaluating each top-level form in the file.

|               |                                      |
|---------------|--------------------------------------|
| Load Function | The function which loads programs    |
| Autoloading   | Functions can be loaded on reference |
| Features      | Module management functions          |

## 1.169 jade.guide/Load Function

### Load Function

-----

- Function: load PROGRAM &optional NO-ERROR NO-PATH NO-SUFFIX  
This function loads the file containing the program called PROGRAM; first the file is located then each top-level form contained by the file is read and evaluated in order.

Each directory named by the variable 'load-path' is searched until the file containing PROGRAM is found. In each directory three different file names are tried,

1. PROGRAM with '.jlc' appended to it. Files with a '.jlc' suffix are usually compiled Lisp files. See  
Compiled Lisp  
.
2. PROGRAM with '.jl' appended, most uncompiled Lisp programs are stored in files with names like this.
3. PROGRAM with no modifications.

If none of these gives a result the next directory is searched in the same way, when all directories in 'load-path' have been exhausted and the file still has not been found an error is signalled.

Next the file is opened for reading and Lisp forms are read from it one at a time, each form is evaluated before the next form is read. When the end of the file is reached the file has been loaded and this function returns 't'.

The optional arguments to this function are used to modify its behaviour,

#### NO-ERROR

When this argument is non-'nil' no error is signalled if the file can not be located. Instead the function returns 'nil'.

#### NO-PATH

The variable 'load-path' is not used, PROGRAM must point to the file from the current working directory.

**NO-SUFFIX**

When non-`'nil'` no `'jlc'` or `'jl'` suffixes are applied to the PROGRAM argument when locating the file.

If a version of the program whose name ends in `'jlc'` is older than a `'jl'` version of the same file (i.e. the source code is newer than the compiled version) a warning is displayed and the `'jl'` version is used.

```
(load "foobar")
error--> File error: Can't open lisp-file, foobar
```

```
(load "foobar" t)
=> nil
```

**- Variable: load-path**

A list of strings, each element is the name of a directory which is prefixed to the name of a program when Lisp program files are being searched for.

```
load-path
=> (" " "/usr/local/lib/jade/3.2/lisp/")
```

The element `'"` means the current directory, note that directory names should have an ending `'/'` (or whatever) so that when concatenated with the name of the file they make a meaningful filename.

**- Variable: lisp-lib-dir**

The name of the directory in which the standard Lisp files are stored.

```
lisp-lib-dir
=> "/usr/local/lib/jade/3.2/lisp/"
```

## 1.170 jade.guide/Autoloading

### Autoloading

-----

Obviously, not all the features of the editor are always used. "Autoloading" allows modules to be loaded when they are referenced. This speeds up the initialisation process and may save memory.

Functions which may be autoloaded have a special form in their symbol's function cell -- an autoload form. This is a list whose first element is the symbol `'autoload'`. When the function call dispatcher finds one of these forms it loads the program file specified in the form then re-evaluates the function call. The true function definition will have been loaded and therefore the call may proceed as normal.

The structure of an autoload form is:

```
(autoload PROGRAM-FILE [IS-COMMAND])
```

PROGRAM-FILE is the argument to give to the 'load' function when the function is to be loaded. It should be the program containing a definition of the autoloaded function.

The optional IS-COMMAND object specifies whether or not the function may be called interactively (i.e. it is an editor command).

- Function: autoload SYMBOL &rest AUTOLOAD-DEFN  
Installs an autoload form into the function cell of the symbol SYMBOL. The form is a cons cell whose car is 'autoload' and whose cdr is the argument AUTOLOAD-DEFN.

Returns the resulting autoload form.

```
(autoload 'foo "foos-file")
=> (autoload "foos-file")
(symbol-function 'foo)
=> (autoload "foos-file")

(autoload 'bar "bars-file" t)
=> (autoload "bars-file" t)
(commandp 'bar)
=> t
```

It is not necessary to call the 'autoload' function manually. Simply prefix the definitions of all the functions which may be autoloaded (i.e. the entry points to your module; \*not\* all the internal functions!) with the magic comment ';;;###autoload'. Then the 'add-autoloads' command can be used to create the necessary calls to the autoload function in the 'autoloads.jl' Lisp file (this file which lives in the Lisp library directory is loaded when the editor is initialised).

'Meta-x add-autoloads'

Scans the current buffer for any autoload definitions. Functions with the comment ';;;###autoload' preceding them have autoload forms inserted into the 'autoloads.jl' file. Simply save this file's buffer and the new autoloads will be used the next time Jade is initialised.

It is also possible to mark arbitrary forms for inclusion in the 'autoloads.jl' file: put them on a single line which starts with the comment ';;;###autoload' call the command.

The unsaved 'autoloads.jl' buffer will become the current buffer.

```
;;;###autoload
(defun foo (bar) ;'foo' is to be autoloaded
 ...

;;;###autoload (setq x y) ;Form to eval on initialisation
```

'Meta-x remove-autoloads'

Remove all autoload forms from the 'autoloads.jl' file which are

marked by the `';;;###autoload'` comment in the current buffer.

The unsaved `'autoloads.jl'` buffer will become the current buffer.

## 1.171 jade.guide/Features

### Features

-----

"Features" correspond to modules of the editor. Each feature is loaded separately. Each feature has a name, when a certain feature is required its user asks for it to be present (with the `'require'` function), the feature may then be used as normal.

When a feature is loaded one of the top-level forms evaluated is a call to the `'provide'` function. This names the feature and installs it into the list of present features.

#### - Variable: features

A list of the features currently present (that is, loaded). Each feature is represented by a symbol. Usually the print name of the symbol (the name of the feature) is the same as the name of the file it was loaded from, minus any `'.jl'` or `'.jlc'` suffix.

```
features
=> (info isearch fill-mode texinfo-mode lisp-mode xc)
```

#### - Function: provide FEATURE

Adds FEATURE (a symbol) to the list of features present. A call to this function is normally one of the top-level forms in a module.

```
;;; maths.jl -- the 'maths' module

(provide 'maths)
...
```

#### - Function: require FEATURE &optional FILE

Show that the caller is planning to use the feature FEATURE (a symbol). This function will check the `'features'` variable to see if FEATURE is already loaded, if so it will return immediately.

If FEATURE is not present it will be loaded. If FILE is non-`'nil'` it specifies the first argument to the `'load'` function, else the print name of the symbol FEATURE is used.

```
;;; physics.jl -- the 'physics' module

(require 'maths) ;Need the 'maths' module
(provide 'physics)
...
```

## 1.172 jade.guide/Compiled Lisp

Compiled Lisp

=====

Jade contains a rudimentary Lisp compiler; this takes a Lisp form or program and compiles it into a "byte-code" form. This byte-code form contains a string of byte instructions, a vector of data constants and some other information.

The main reason for compiling your programs is to increase their speed, it is difficult to quantify the speed increase gained -- some programs (especially those using a lot of macros) will execute many times quicker than their uncompiled version whereas others may only execute a bit quicker.

Compilation Functions

How to compile Lisp programs

Compilation Tips

Getting the most out of the compiler

Disassembly

Examining compiled functions

## 1.173 jade.guide/Compilation Functions

Compilation Functions

-----

- Function: `compile-form` FORM

This function compiles the Lisp form FORM into a byte-code form which is returned.

```
(compile-form '(setq foo bar))
=> (jade-byte-code "?F!" [bar foo] 2)
```

- Command: `compile-file` FILE-NAME

This function compiles the file called FILE-NAME into a file of compiled Lisp forms whose name is FILE-NAME with 'c' appended to it (i.e. if FILE-NAME is 'foo.jl' it will be compiled to 'foo.jlc').

If an error occurs while the file is being compiled any semi-written file will be deleted.

When called interactively this function will ask for the value of FILE-NAME.

- Command: `compile-directory` DIRECTORY &optional FORCE EXCLUDE

Compiles all the Lisp files in the directory called DIRECTORY which either haven't been compiled or whose compiled version is older than the source file (Lisp files are those ending in '.jl').

If the optional argument FORCE is non-'nil' \*all\* Lisp files will be recompiled whatever the status of their compiled version.

The EXCLUDE argument may be a list of filenames, these files will \*not\* be compiled.

When this function is called interactively it prompts for the directory.

- Function: compile-lisp-lib &optional FORCE  
Uses 'compile-directory' to compile the library of standard Lisp files. If FORCE is non-'nil' all of these files will be compiled.

The 'autoloads.jl' is \*never\* compiled since it is often modified and wouldn't really benefit from compilation anyway.

- Function: jade-byte-code BYTE-CODES CONSTANTS MAX-STACK  
Interprets the string of byte instructions BYTE-CODES with the vector of constants CONSTANTS. MAX-STACK defines the maximum number of stack cells required to interpret the code.

This function is \*never\* called by hand. The compiler will produce calls to this function when it compiles a form or a function.

```
(setq x 1
 y 3)
=> 3
(setq comp (compile-form '(cons x y)))
=> (jade-byte-code "??K" [x y] 2)
(eval comp)
=> (1 . 3)
```

## 1.174 jade.guide/Compilation Tips

### Compilation Tips

---

Here are some tips for making compiled code run fast:

- \* Always favour iteration over recursion; function calls are relatively slow. The compiler doesn't know about tail recursion or whatever so you'll have to do this explicitly.

For example, the most elegant way of searching a list is to use recursion,

```
(defun scan-list (list elt)
 "Search the LIST for an element ELT. Return it if one is found."
 (if (eq (car list) elt)
 elt
```



```
(scan-list (cdr list) elt))
```

but this is fairly slow. Instead, iterate through each element,

```
(defun scan-list (list elt)
 (while (consp list)
 (when (eq (car list) elt)
 (return elt))
 (setq list (cdr list))))
```

- \* In some cases the functions 'member', 'memq', 'assoc', etc... can be used to search lists. Since these are primitives written in C they will run \*much\* faster than an equivalent Lisp function.

So the above 'scan-list' example can be rewritten as,

```
(defun scan-list (list elt)
 (car (memq elt list)))
```

Also note that the 'mapcar' and 'mapc' functions are useful (and efficient) when using lists.

- \* Whenever possible use the 'when' and 'unless' conditional structures; they are more efficient than 'cond' or 'if'.
- \* Careful use of named constants (see Constant Variables) can increase the speed of some programs. For example, in the Lisp compiler itself all the opcode values (small integers) are defined as constants.

I must stress that in some cases constants are \*not\* suitable; they may drastically increase the size of the compiled program (when the constants are 'big' objects, i.e. long lists) or even introduce subtle bugs (since two references to the same constant may not be 'eq' whereas two references to the same variable are always 'eq').

- \* Many primitives have corresponding byte-code instructions; these primitives will be quicker to call than those that don't (and incur a normal function call). Currently, the functions which have byte-code instructions (apart from all the special forms) are:

```
'cons', 'car', 'cdr', 'rplaca', 'rplacd', 'nth', 'nthcdr', 'aset',
'aref', 'length', 'eval', '+', '*', '/', '%', 'lognot', 'not',
'logior', 'logand', 'equal', 'eq', '=', '/=', '>', '<', '>=',
'<=', 'l+', 'l-', '-', 'set', 'fset', 'lsh', 'zerop', 'null',
'atom', 'consp', 'listp', 'numberp', 'stringp', 'vectorp', 'throw',
'fboundp', 'boundp', 'symbolp', 'get', 'put', 'signal', 'return',
'reverse', 'nreverse', 'assoc', 'assq', 'rassoc', 'rassq', 'last',
'mapcar', 'mapc', 'member', 'memq', 'delete', 'delq', 'delete-if',
'delete-if-not', 'copy-sequence', 'sequencep', 'functionp',
'special-formp', 'subrp', 'eql', 'set-current-buffer',
'current-buffer', 'bufferp', 'markp', 'windowp'.
```

- \* When a file is being compiled each top-level form it contains is

inspected to see if it should be compiled into a byte-code form. Different types of form are processed in different ways:

- \* Function and macro definitions have their body forms compiled into a single byte-code form. The doc-string and interactive declaration are not compiled.
- \* Calls to the 'require' function are evaluated then the unevaluated form is written as-is to the output file. The reason it is evaluated is so that any macros defined in the required module are loaded before they are called by the program being compiled.
- \* If the form is a list form (see  
List Forms  
) and the symbol  
which is the car of the list is one of:

```
'if', 'cond', 'when', 'unless', 'let', 'let*', 'catch',
'unwind-protect', 'error-protect', 'with-buffer',
'with-window', 'progn', 'progl', 'prog2', 'while', 'and',
'or'.
```

then the form is compiled. Otherwise it is just written to the output file in its uncompiled state.

If your program contains a lot of top-level forms which you know will not be compiled automatically, consider putting them in a 'progn' block to make the compiler coalesce them into one byte-code form.

## 1.175 jade.guide/Disassembly

Disassembly

-----

It is possible to disassemble byte-code forms; originally this was so I could figure out why the compiler wasn't working but if you're curious about how the compiler compiles a form it may be of use to you.

Naturally, the output of the disassembler is a listing in Jade's pseudo-machine language -- it won't take a byte-code form and produce the equivalent Lisp code!

- Command: disassemble-fun FUNCTION &optional STREAM

This function disassembles the compile Lisp function FUNCTION. It writes a listing to the output stream STREAM (normally the value of the 'standard-output' variable).

When called interactively it will prompt for a function to disassemble.

When reading the output of the disassembler bear in mind that Jade

simulates a stack machine for the code to run on. All calculations are performed on the stack, the value left on the stack when the piece of code ends is the value of the byte-code form.

## 1.176 jade.guide/Hooks

### Hooks

=====

A "hook" allows you to wedge your own pieces of Lisp code into the editor's operations. These pieces of code are evaluated via the hook and the result is available to the hook's caller.

### Functions As Hooks

Some hooks are a single function,

### Normal Hooks

Others may be a list of pieces of code to evaluate.

### Standard Hooks

A table of the predefined hooks

## 1.177 jade.guide/Functions As Hooks

### Functions As Hooks

-----

Some hooks only allow a single piece of code to be hooked in. Usually a normally-undefined function is used; to install your hook defined a function with the name of the hook. When the hook is to be evaluated the function is called.

Generally the name of the hook's function will end in '-function'.

An alternative scheme is to use a variable to store the hook, its value should be the function to call.

## 1.178 jade.guide/Normal Hooks

### Normal Hooks

-----

This is the standard type of hook, it is a variable whose value is a

---

list of functions. When the hook is evaluated each of the named functions will be called in turn until one of them returns a value which is not `'nil'`. This value becomes the value of the hook and no more of the functions are called. If all of the functions in the hook return `'nil'` the value of the hook is `'nil'`.

The names of hooks of this type will normally end in `'-hook'`.

- Function: `add-hook HOOK FUNCTION &optional AT-END`  
This function adds a new function `FUNCTION` to the list of functions installed in the (list) hook `HOOK` (a symbol).

If `AT-END` is non-`'nil'` the new function is added at the end of the hook's list of functions (and therefore will be called last when the hook is evaluated), otherwise the new function is added to the front of the list.

```
text-mode-hook
=> (fill-mode-on)
(add-hook 'text-mode-hook 'my-function)
=> (my-function fill-mode-on)
```

- Function: `remove-hook HOOK FUNCTION`  
This function removes the function `FUNCTION` from the list of functions stored in the (list) hook `HOOK` (a symbol).

\*All\* instances of `FUNCTION` are deleted from the hook.

```
text-mode-hook
=> (my-function fill-mode-on)
(remove-hook 'text-mode-hook 'my-function)
=> (fill-mode-on)
```

- Function: `eval-hook HOOK &rest ARGS`  
Evaluates the (list) hook `HOOK` (a symbol) with argument values `ARGS`.

Each function stored in the hook is applied to the `ARGS` in turn until one returns non-`'nil'`. This non-`'nil'` value becomes the result of the hook. If all functions return `'nil'` then the result of the hook is `'nil'`.

Note that most functions which are installed in hooks should always return `'nil'` to ensure that all the functions in the hook are evaluated.

## 1.179 jade.guide/Standard Hooks

### Standard Hooks

---

This is a table of the predefined hooks in Jade:

```
'asm-cpp-mode-hook'
See
```

---

Asm mode

.

`'asm-mode-hook'`

See

Asm mode

.

`'auto-save-hook'`

See

Controlling Auto-Saves

.

`'buffer-menu-mode-hook'`

`'c-mode-hook'`

See

C mode

.

`'destroy-window-hook'`

See

Closing Windows

.

`'gdb-hook'`

`'idle-hook'`

See

Idle Actions

.

`'indented-text-mode-hook'`

See

Indented-Text mode

.

`'insert-file-hook'`

See

Reading Files Into Buffers

.

`'kill-buffer-hook'`

See

Destroying Buffers

.

`'lisp-mode-hook'`

See

Lisp mode

.

`'make-window-hook'`

See

Opening Windows

.

`'open-file-hook'`

See

Reading Files Into Buffers

.

`'read-file-hook'`

See

Reading Files Into Buffers

.

`'shell-callback-function'`

`'shell-mode-hook'`

`'texinfo-mode-hook'`

See

Texinfo mode

.

`'text-mode-hook'`

See

Text mode

.

`'unbound-key-hook'`

See

Event Loop

.

`'window-closed-hook'`

See

Event Loop

.

`'write-file-hook'`

See

Writing Buffers

.

## 1.180 jade.guide/Buffers

Buffers

=====

A "buffer" is a Lisp object containing a 'space' in which files (or any pieces of text) may be edited, either directly by the user or by Lisp programs.

Each window (see

Windows

) may display any one buffer at any time, the

buffer being displayed by the current window is known as the "current buffer". This is the buffer which functions will operate on by default.

- Function: `bufferp` OBJECT

Returns 't' if its argument is a buffer.

|                          |                                    |
|--------------------------|------------------------------------|
| Buffer Attributes        | Data contained in a buffer object  |
| Creating Buffers         | How to create empty buffers        |
| Modifications to Buffers | Is a buffer modified?              |
| Read-Only Buffers        | Unmodifiable buffers               |
| Destroying Buffers       | Deleting a buffer and its contents |
| Special Buffers          | Program-controlled buffers         |
| The Buffer List          | Each window has a list of buffers  |
| The Current Buffer       | One buffer is the default buffer   |

## 1.181 jade.guide/Buffer Attributes

### Buffer Attributes

---

All buffer objects store a set of basic attributes, some of these are:

#### "name"

Each buffer has a unique name.

- Function: `buffer-name` &optional BUFFER  
Returns the name of the buffer BUFFER, or of the current buffer if BUFFER is undefined.

```
(buffer-name)
=> "programmer.texi"
```

- Function: `set-buffer-name` NAME &optional BUFFER  
Sets the name of the buffer BUFFER (or the current buffer) to the string NAME.

Note that NAME is not checked for uniqueness, use the `'make-buffer-name'` function if you want a guaranteed unique name.

- Function: `make-buffer-name` NAME
-

Returns a unique version of the string NAME so that no existing buffer has the same string as its name. If a clash occurs a suffix '<N>' is appended to NAME, where N is the first number which guarantees the uniqueness of the result.

- Function: `get-buffer NAME`  
Returns the existing buffer whose name is NAME, or 'nil' if no such buffer exists.

#### "file name"

Since buffers often contain text belonging to files on disk the buffer stores the name of the file its text was read from. See

Editing Files

.

- Function: `buffer-file-name &optional BUFFER`  
Returns the name of the file stored in BUFFER. If no file is stored in the buffer the null string ('') is returned.

```
(buffer-file-name)
=> "man/programmer.texi"
```

- Function: `set-buffer-file-name NAME &optional BUFFER`  
This function sets the file-name of the buffer to the string NAME.
- Function: `get-file-buffer FILE-NAME`  
Searches for an existing buffer containing the file FILE-NAME then returns it, or 'nil' if no such buffer exists.

#### "contents"

The contents of a buffer is the text it holds. This is stored as an array of lines. See

Text

.

#### "tab size"

This is the spacing of tab stops. When the contents of the buffer is being displayed (in a window) this value is used.

- Variable: `tab-size`  
A buffer-local variable which holds the size of tab stops in the buffer.

#### "glyph table"

Each buffer has its own glyph table which is used when the buffer is being displayed. See

Buffer Glyph Tables

.

#### "local variables"

Each buffer can have its own value for any variable, these local values are stored in an alist which lives in the buffer object. See

Buffer-Local Variables

.



- Function: `buffer-variables` &optional BUFFER  
Returns the alist of local variables in the buffer. Each alist element is structured like, `'(SYMBOL . LOCAL-VALUE)'`.

#### "modification counter"

Each modification made to the buffer increments its modification counter. See  
[Modifications to Buffers](#)  
 .

- Function: `buffer-changes` &optional BUFFER  
Returns the number of modifications made to the buffer since it was created.

#### "undo information"

When a modification is made to a buffer enough information is recorded so that the modification can later be undone. See

[Controlling Undo](#)  
 .

All other buffer-specific information is kept in buffer-local variables.

## 1.182 jade.guide/Creating Buffers

### Creating Buffers

-----

- Function: `make-buffer` NAME  
Creates and returns a new buffer object. Its name will be a unique version of NAME (created by the `'make-buffer-name'` function).

The buffer will be totally empty and all its attributes will have standard values.

```
(make-buffer "foo")
=> #<buffer foo>
```

- Function: `open-buffer` NAME  
If no buffer called NAME exists, creates a new buffer of that name and adds it to the end of each windows `'buffer-list'`. This function always returns the buffer called NAME.

For more ways of creating buffers see  
[Editing Files](#)  
 .

## 1.183 jade.guide/Modifications to Buffers

### Modifications to Buffers

-----

Each buffer maintains a counter which is incremented each time the contents of the buffer is modified. It also holds the value of this counter when the buffer was last saved, when the two numbers are different the buffer is classed as have being "modified".

- Function: `buffer-modified-p` &optional BUFFER  
This function returns `'t'` when the buffer has been modified.
- Function: `set-buffer-modified` BUFFER STATUS  
Sets the modified status of the buffer BUFFER. When STATUS is `'nil'` the buffer will appear to be unmodified, otherwise it will look modified.

## 1.184 jade.guide/Read-Only Buffers

### Read-Only Buffers

-----

When a buffer has been marked as being read-only no modifications may be made to its contents (neither by the user nor a Lisp program).

- Function: `buffer-read-only-p` &optional BUFFER  
Returns `'t'` when the buffer is read-only.
- Function: `set-buffer-read-only` BUFFER READ-ONLY  
When READ-ONLY is non-`'nil'` the buffer BUFFER is marked as being read-only, otherwise it is read-write.
- Variable: `inhibit-read-only`  
When this variable is non-`'nil'` any buffer may be modified, even if it is marked as being read-only.

Lisp programs can temporarily bind a non-`'nil'` value to this variable when they want to edit one of their normally read-only buffers.

## 1.185 jade.guide/Destroying Buffers

### Destroying Buffers

-----

Since all Lisp objects have indefinite extent (i.e. they live until there are no references to them) a buffer will be automatically destroyed when all references to it disappear.

---

Alternatively one of the following functions can be used to explicitly kill a buffer; the buffer object will still exist but all data associated with it (including the text it contains) will be released.

- Command: `kill-buffer BUFFER`  
Removes the buffer `BUFFER` (a buffer or the name of a buffer) from all windows (any windows displaying `BUFFER` will be changed to display the previous buffer they showed) and destroys the buffer.

The hook `'kill-buffer-hook'` is evaluated before the buffer is killed with `BUFFER` as its argument.

If the buffer contains unsaved modifications the user will be asked if they really want to lose them before the buffer is killed (if the answer is yes).

When called interactively a buffer will be prompted for.

- Hook: `kill-buffer-hook`  
Hook called by `'kill-buffer'` before it does anything. If a function in the hook doesn't want the buffer deleted it should signal some sort of error.
- Function: `destroy-buffer BUFFER`  
This function may be used to remove all data stored in the buffer object manually. Also, any marks in this buffer are made non-resident.

After applying this function to a buffer the buffer will contain one empty line.

Use this function wisely, there are no safety measures taken to ensure valuable data is not lost.

## 1.186 jade.guide/Special Buffers

### Special Buffers

-----

When a buffer is "special" it means that it is controlled by a Lisp program, not by the user typing into it (although this can happen as well).

Special buffers are used for things like the `'*jade*'` or `'*Info*'` buffers (in fact most of the buffers whose names are surrounded by asterisks are special).

What the special attribute actually does is make sure that the buffer is never truly killed (`'kill-buffer'` removes it from each window's `'buffer-list'` but doesn't call `'destroy-buffer'` on it) and modifications don't cause the `'+'` flag to appear in the status line.

- Function: `buffer-special-p &optional BUFFER`
-

Returns `'t'` if the buffer is marked as being special.

- **Function:** `set-buffer-special BUFFER SPECIAL`  
Sets the value of the special flag in the buffer `BUFFER` to the value of `SPECIAL` (`'nil'` means non-special, anything else means special).

Another type of special buffer exists; the "mildly-special buffer".

- **Variable:** `mildly-special-buffer`  
When this buffer-local variable is set to `'t'` (it is `'nil'` by default) and the buffer is marked as being special, the `'kill-buffer'` function is allowed to totally destroy the buffer.

## 1.187 jade.guide/The Buffer List

### The Buffer List

Each window (see `Windows`) has a list of buffers which may be displayed in that window. It is arranged in "most-recently-used" order, so that the car of the list is the buffer currently being shown in the window, the second element the window previously being shown and so on.

- **Variable:** `buffer-list`  
A variable, local to each window, which contains a list of the buffers available in the window. The list is maintained in most-recently-used order.

```
buffer-list
=> (#<buffer programmer.texi> #<buffer *Help*>
 #<buffer buffers.c> #<buffer buffers.jl>
 #<buffer edit.c> #<buffer edit.h>
 #<buffer *jade*> #<buffer lisp.jl>
 #<buffer *compilation*> #<buffer *Info*>)
```

Generally each window's `'buffer-list'` contains the same buffers, each window has its own value for the variable so it can be kept in the correct order (each window will probably be displaying different buffers).

- **Function:** `add-buffer BUFFER`  
This function ensures that the buffer `BUFFER` is in each window's `'buffer-list'`. If it isn't it is appended to the end of the list.
- **Function:** `remove-buffer BUFFER`  
Deletes all references to `BUFFER` in each window's `'buffer-list'`.
- **Command:** `bury-buffer &optional BUFFER ALL-WINDOWS`  
Puts `BUFFER` (or the currently displayed buffer) at the end of the current window's `'buffer-list'` then switch to the buffer at the head of the list.

If ALL-WINDOWS is non-`'nil'` this is done in all windows (the same buffer will be buried in each window though).

- Command: `rotate-buffers-forward`  
 Moves the buffer at the head of the `'buffer-list'` to be last in the list, the new head of the `'buffer-list'` is displayed in the current window.

## 1.188 jade.guide/The Current Buffer

### The Current Buffer

-----

The "current buffer" is the buffer being displayed in the current window (see

`Windows`

), all functions which take an optional `BUFFER` argument will operate on the current buffer if this argument is undefined. Similarly if a `WINDOW` argument to a function is left undefined the current window will be used.

- Function: `current-buffer &optional WINDOW`  
 Returns the buffer being displayed by the window `WINDOW` (or the current window).

```
(current-buffer)
=> #<buffer programmer.texi>
```

The `'set-current-buffer'` function sets the current buffer of a window. If, when the window is next redisplayed (i.e. after each command), the current buffer is different to what it was at the last redisplay the new buffer will be displayed in the window.

- Function: `set-current-buffer BUFFER &optional WINDOW`  
 Sets the buffer that the window is displaying.

Usually a window's current buffer will be the buffer which is at the head of the window's `'buffer-list'`. The function `'goto-buffer'` can be used to set both of these at once.

- Function: `goto-buffer BUFFER`  
 Set the current buffer to `BUFFER` which is either a buffer or a string naming a buffer. The selected buffer is moved to the head of the window's `'buffer-list'`.

If `BUFFER` is a string and no buffer exists of that name a new one is created.

Often you will want to temporarily switch to a different current buffer, that is what the `'with-buffer'` special form is for.

- Special Form: `with-buffer BUFFER FORMS...`
-

Temporarily sets the current buffer to the value of evaluating BUFFER, then evaluates the FORMS in sequence. The old value of the current buffer is reinstated and the structure returns the value of the last of the FORMS to be evaluated.

If the implicit progn evaluating FORMS is exited abnormally the old value of the current buffer will still be reinstated.

If the window is redisplayed while the FORMS are being evaluated (i.e. in a recursive edit) the new buffer will be drawn into the window.

```
(with-buffer new-buffer ;Enter a recursive edit in
 (recursive-edit)) ; the buffer 'new-buffer'.
```

## 1.189 jade.guide/Windows

Windows

=====

A "window" is a Lisp object representing a window (a rectangular section of the display) open in the windowing-system you are running Jade in.

Windows have two main functions, firstly to provide a means of seeing the contents of a buffer and secondly to receive input events. For more details about event handling see

Event Loop

.

A window *\*always\** displays a buffer and there is *\*always\** at least one window open. The editor remembers which of the open windows is the "current window", this is normally the window it last received an input event from, though it can be set by programs.

For some basic details about using windows see

Using Windows

.

- Function: windowp OBJECT

This function returns 't' if its argument is a window.

- Variable: window-list

This variable's value is a list of all the currently open windows. The order of the elements in the list is insignificant.

```
window-list
```

```
=> (#<window 20971528 *Info*> #<window 20971524 *jade*>)
```

Opening Windows

Creating new windows

|                                       |  |
|---------------------------------------|--|
| Closing Windows                       |  |
| Deleting windows                      |  |
| Iconifying Windows                    |  |
| Temporarily removing windows          |  |
| Displaying Messages                   |  |
| Messages to the user                  |  |
| The Current Window                    |  |
| The activated window, used by default |  |
| Window Font                           |  |
| Each window may use a different font  |  |
| Window Information                    |  |
| Details of a window's current state   |  |
| Rendering                             |  |
| How buffers are drawn in windows      |  |
| Block Marking                         |  |
| Highlighting a region of a window     |  |

## 1.190 jade.guide/Opening Windows

### Opening Windows

-----

- Function: open-window &optional BUFFER X Y WIDTH HEIGHT

Opens a new window and returns it. If BUFFER is defined it is the buffer to display in the new window, otherwise the current buffer is displayed.

The X and Y arguments are the pixel coordinates of the new window's top left corner in the display. The WIDTH and HEIGHT arguments are the size of the window in columns and rows of characters respectively.

What happens when the position and size of the window is undefined will depend on the underlying window system, on the Amiga the window will probably be the same as the current window, in X11 the window manager will probably let the user size it interactively.

The new window will have its 'buffer-list' variable initialised suitably and it will be added to the head of the 'window-list' variable.

The 'make-window' function is the lowest level of creating a new window, 'open-window' uses it to open the window.

- Function: make-window &optional X Y WIDTH HEIGHT

---

Creates a new window and returns it, the arguments are similar to those of the same name in the 'open-window' function. The window will display the current buffer.

After the window is created the 'make-window-hook' will be called with the window as its argument.

- Hook: make-window-hook  
Hook called each time a new window is created. It has one argument, the new window.
- Variable: pub-screen  
This window-local variable is only used on the Amiga version of Jade; it holds the name of the public screen which windows are opened on. By default this is the Workbench screen.

When a window is opened it inherits this value from the current window at the time.

## 1.191 jade.guide/Closing Windows

### Closing Windows

-----

Unlike buffers, window objects don't have indefinite extent, even when a window is incapable of being referenced the object will not be destroyed by the garbage collector; count the user looking at the window as a reference!

When the window is closed (by the 'destroy-window' function) the object loses its 'window-ness' and the garbage collector is free to reclaim its memory.

- Function: close-window &optional WINDOW  
This function closes the window WINDOW (or the current window) and deletes its entry from the 'window-list' variable.

If this window is the only one the editor has open the user is asked if it's okay to lose any modified buffers before the window is closed.

- Function: close-other-windows &optional WINDOW  
Uses 'close-window' to close all windows except WINDOW (or the current window).
- Function: destroy-window WINDOW  
Closes the window WINDOW. After a window object has been closed it is no longer a member of the type 'window'.

Before closing the window the 'destroy-window-hook' is evaluated with the window being destroyed as an argument.

When the last window is closed the editor will exit automatically.

---



Like the 'destroy-buffer' function, this function is dangerous if used carelessly.

Both 'close-window' and 'close-other-windows' eventually call this function.

- Hook: destroy-window-hook  
Hook called by 'destroy-window' before it does anything. It has one argument -- the window to be destroyed.

## 1.192 jade.guide/Iconifying Windows

### Iconifying Windows

-----

When you don't want a window cluttering the display, but don't want to kill it totally it can be iconified; the window will be displayed as a small icon which can be reactivated when the window is wanted again.

- Function: sleep-window &optional WINDOW  
Iconifies the specified window.
- Function: unsleep-window &optional WINDOW  
Uniconifies the specified window. This may be done automatically if the user needs to be prompted.
- Function: toggle-iconic  
Toggles the current window between the iconified and normal states. This command is bound to the key sequence 'Ctrl-z'.
- Function: window-asleep-p  
Returns 't' when the current window is iconified.

## 1.193 jade.guide/Displaying Messages

### Displaying Messages

-----

Often it is useful to be able to show the user a short one-line message, this is what the 'message' function does.

- Function: message MESSAGE &optional DISPLAY-NOW  
This function displays the string MESSAGE in the status line of the current window, then returns MESSAGE.

If DISPLAY-NOW is non-'nil' the message is rendered into the window immediately, otherwise it will not be visible until the next general redisplay (usually after each command exits).

Note that an alternate way of writing in the status line is to use

---

the output stream `'t'`. See  
     Output Streams

.

When writing interactive programs it is sometimes useful to be able to render the cursor in the status line. This shows that the next key press will not be subject to normal editing key bindings but to the special user interface (usually explained by a message in the status line).

For example the `'y-or-n-p'` function uses this technique to show that it needs an answer.

- Variable: `status-line-cursor`  
     When this window-local variable is non-`'nil'` the window's cursor is rendered at the end of the message in the status line, not at the cursor's position in the main display.

Another way of alerting the user is to use the `'beep'` function,

- Function: `beep`  
     This function rings a bell or flashes the current window or screen depending on your system.

## 1.194 jade.guide/The Current Window

The Current Window

-----

The current window is the window that functions operate on by default; every time the event loop receives some input from the user the window which the input event originated in becomes the current window. It is also possible for Lisp programs to set the current window, either permanently or temporarily.

The "active window" is the window which the windowing system will send any keyboard input to. Since Jade sets the current window to where it receives input from, it is often the case that the current window is the same as the active window. Jade also provides the means to set the active window; in some cases this may be best left to the user though.

- Function: `current-window`  
     This function returns the current window.

```
(current-window)
=> #<window 20971524 programmer.texi>
```

- Function: `set-current-window WINDOW &optional ACTIVATE`  
     This function sets the current window to be the window `WINDOW`. If the optional argument `ACTIVATE` is non-`'nil'` this window will also become the active window.

When using the `ACTIVATE` argument bear in mind that it may be

confusing for the user if the active window is suddenly changed; only change the active window synchronously with some input from the user.

- Special Form: with-window WINDOW FORMS...  
Temporarily sets the current window to the value of evaluating the form WINDOW, then uses an implicit progn to evaluate the FORMS. The old current window is then reinstated before returning the value of the implicit progn.

## 1.195 jade.guide/Window Font

### Window Font

-----

Each window may use a different font; this font will be used for rendering all text in the window. When windows are created they inherit their font from the current window at the time.

Currently Jade only allows the use of fixed-width fonts; proportional fonts won't work properly.

- Command: set-font FONT-NAME &optional WINDOW  
This function sets the font used in the window WINDOW (or the current window) to the font named by the string FONT-NAME.

The format of the string FONT-NAME depends on the underlying windowing system:

#### X11

Simply use the standard name of the font, asterisk characters work like usual (i.e. match zero or more characters).

#### Amiga

This is different to the normal Amiga conventions, use the name of the font followed by a dash and then the size of the font. For example to get an 8-point topaz font, use `'topaz.font-8'`.

When this function is called interactively it will prompt for FONT-NAME.

- Function: font-name &optional WINDOW  
Returns the name of the font being used in the specified window.

Note that on an Amiga this will only return the name, and not the size of the font. For example, if `'set-font'` has been used with an argument of `'"topaz.font-8"'`, a call to `'font-name'` would produce `'"topaz.font"'`.

- Function: font-x-size &optional WINDOW  
Returns the width (in pixels) of a character in the specified window's font.

```
(font-x-size)
=> 7
```

- Function: font-y-size &optional WINDOW  
Returns the height in pixels of each character in the window's font.

```
(font-y-size)
=> 13
```

## 1.196 jade.guide/Window Information

### Window Information

-----

There are a number of functions which provide information about the current state of a window.

- Function: window-id &optional WINDOW  
Returns an integer which is the window system's 'handle' on the window WINDOW (or the current window). Under X11 this is the Window identifier, on an Amiga it's a pointer to the window's 'struct Window'.

```
(window-id)
=> 20971524
```

- Function: window-count  
Returns the number of currently-opened windows.
  - Function: screen-width  
Returns the width of the root window or screen in pixels.
  - Function: screen-height  
Returns the height in pixels of the root window.
  - Function: window-left-edge  
Returns the x coordinate of the current window relative to the root window's top-left corner.
  - Function: window-top-edge  
Returns the y coordinate of the current window relative to the root window's top-left corner.
  - Function: window-width  
Returns the width, in pixels, of the current window.
  - Function: window-height  
Returns the height in pixels of the current window.
  - Function: window-bar-height  
Only used by Amigas, this returns the height of the current window's title bar. This will always be zero in X.
-

- Function: `screen-top-line`  
Returns the line number of the first line being shown in the current window.
- Function: `screen-bottom-line`  
Returns the line number of the last line being shown in the current window.
- Function: `screen-first-column`  
Returns the column number of the first column being shown in the current window.
- Function: `screen-last-column`  
Returns the column number of the last column being shown in the current window.

## 1.197 jade.guide/Rendering

### Rendering

-----

After each command is executed a full redisplay is done; the display of each window is made to be consistent with the contents of the buffer it is showing.

- Function: `refresh-all`  
This function calls the redisplay code, any windows, whose display is inconsistent with what it should be displaying, are updated.
  - Function: `cursor ON`  
Turns the cursor in the current window on or off (depending on whether ON is non-`'nil'` or not). Normally the cursor is erased while Lisp programs are executing.  
  
If you use this function be sure to leave the cursor undrawn when you've finished.
  - Function: `centre-display &optional WINDOW`  
If possible, this function will arrange it so that the line which the cursor is on (see  
    The Cursor Position  
    ) will be in the centre  
of the display.
  - Function: `next-screen &optional COUNT`  
Move COUNT (or 1 by default) screens forwards in the display, Lisp programs shouldn't need to call this.
  - Function: `prev-screen &optional COUNT`  
Move COUNT screens backwards in the display. Don't call this from Lisp programs.
  - Function: `flush-output`  
This function forces any locally-cached rendering operations to be
-

drawn into the actual window. This should be called after any use of the 'refresh-all' or 'cursor' functions.

Currently this function only actually does anything in the X11 version of Jade (it calls XFlush()), but to ensure the portability of Lisp programs it should be used anyway.

- Variable: max-scroll

This window-local variable defines the maximum number of lines which may be scrolled in one go; if more than this number of lines have to be moved when a redisplay happens the whole window will be redrawn.

- Variable: y-scroll-step-ratio

This window-local variable controls the actual number of lines scrolled when the cursor moves out of the visible part of the window. The number of lines to move the display origin is calculated with the formula:

```
(/ TOTAL-LINES-IN-WINDOW y-scroll-step-ratio)
```

If the variable's value is zero then the window will be scrolled by the least number of lines necessary to get the cursor back into the visible part.

- Variable: x-scroll-step-ratio

Similar to 'y-scroll-step-ratio', except that it's used when the cursor disappears to the left or the right of the display.

## 1.198 jade.guide/Block Marking

### Block Marking

-----

Each window may define one "block", this is a region of the buffer displayed in the window which is rendered in the opposite colours to normal (i.e. the same as the normal cursor, when the cursor is in a block it's drawn in the inverse of the block). Blocks are primarily used for marking areas of a buffer which will subsequently be manipulated.

Normally the area of the buffer contained by a block is delimited by two positions; the start and end of the block (these will track changes made to the buffer and adjust themselves, like marks do). It is also possible to mark rectangular blocks; these are also delimited by two positions, but they define the two opposite corners of the rectangular block.

- Function: blockp

Returns 't' if a block is marked in the current window.

- Function: mark-block START-POS END-POS

Define the beginning and end markers of the block to display in the

---

current window.

- Command: block-kill  
Unmark the block displayed in the current window.
- Command: mark-word COUNT &optional POS  
Mark COUNT words from POS (or the cursor pos) in the current window.
- Command: mark-whole-buffer  
Mark the whole of the current buffer.
- Function: block-start  
Returns the position of the beginning of the block marked in the current window. If no block is defined returns 'nil'.
- Function: block-end  
Returns the position of the end of the block, or 'nil' if no block is defined in the current window.
- Command: block-toggle  
Toggles between marking the beginning, marking the end and totally unmarking the block in the current window.
- Function: rect-blocks-p &optional WINDOW  
Returns 't' if the block marked in the window is drawn as a rectangle.
- Function: set-rect-blocks WINDOW STATUS  
Defines whether or not the block drawn in WINDOW is drawn as a rectangle or not. If STATUS is 'nil' it isn't.
- Command: toggle-rect-blocks  
Toggles between marking normal and rectangular blocks in the current window.

## 1.199 jade.guide/Positions

Positions

=====

A "position" is a Lisp object representing the location of one of the characters in the contents of a buffer (see

Buffers

). Since Jade stores

buffer contents as an array of lines, two index values are needed to reference a single character. A position object contains two integers; the column and line numbers of the character, both these values count upwards from zero (i.e. the first character in a buffer has line and column numbers of zero).

Position objects have no read syntax; they print as,

```
#<pos COLUMN LINE>
```

- Function: `posp OBJECT`  
This function returns `'t'` when its argument is a position object.
- Function: `pos COLUMN LINE`  
Creates and returns a new position object, it points to column number `COLUMN` and line number `LINE` (both integers).
- Function: `copy-pos POS`  
Creates a new copy of the position object `POS`.

#### Position Components

Accessing the members of a position

#### The Cursor Position

Where the cursor is drawn in the display

#### Movement Functions

Position-motion functions

#### Positions and Offsets

Converting between positions and buffer offsets

## 1.200 jade.guide/Position Components

### Position Components

As previously noted, each position object has two components; one number defining the column, the other defining the line that the position represents. These components can be accessed individually.

- Function: `pos-col POS`  
Returns the column which the position object `POS` points to.

```
(pos-col (pos 1 2))
=> 1
```

- Function: `pos-line POS`  
This function returns the line number which `POS` points to.

- Function: `set-pos-col POS NEW-COL`  
Sets the number of the column which the position object `POS` points to, to `NEW-COL` (an integer), then returns `col`.

```
(setq x (pos 1 2))
=> #<pos 1 2>
(set-pos-col x 3)
=> 3
```

x



```
=> #<pos 3 2>
```

- Function: `set-pos-line POS NEW-LINE`  
Similar to `'set-pos-col'` except the line number is modified.

## 1.201 jade.guide/The Cursor Position

### The Cursor Position

-----

Each window displays a "cursor", this is rendered as a character in the opposite colour to what it would usually be (i.e. normally a dark rectangle). The cursor is used to show the user where any characters they type will be inserted, each window has a separate cursor position and buffers which are not being displayed 'remember' the last position of their cursor.

- Function: `cursor-pos`  
This function returns a copy of the cursor position in the current window.

```
(cursor-pos)
=> #<pos 14 5638>
```

- Function: `goto-char POS`  
Sets the position of the current window's cursor to the position object POS, then returns POS.

Note that the components of POS are *\*copied\**, any subsequent modification of POS will not affect the cursor.

If the line number of POS points to a non-existent line the cursor won't be moved and `'nil'` will be returned.

## 1.202 jade.guide/Movement Functions

### Movement Functions

-----

This section documents the functions which are used to create and modify position objects so that they point to a different position which is related to the original position in some way.

The functions which begin `'goto-'` set the cursor position of the current window to the new position; the others do *\*not\** move the cursor, they simply calculate the new position and return it.

In some cases the position argument itself will be modified and returned, this may cause confusion; if there are existing references to the object they subtle bugs may result. Consider the following,

---

```
(setq x (cursor-pos)
 y (next-char 1 x))
```

At first glance this looks as though the variable 'y' will point to one character after the variable 'x' does. Since the 'next-char' function \*modifies\* its argument position \*both\* variables will contain the same object, and therefore, point to the same position.

A solution is,

```
(setq x (cursor-pos)
 y (next-char 1 (copy-pos x)))
```

Read each function's description carefully to see if it alters its arguments!

```
Buffer Extremes
 The edges of a buffer

Character Movement
 Moving in terms of characters,

Word Movement
 or maybe words,

Tab Movement
 tabs,

Line Movement
 lines,

Expression Movement
 or even expressions.
```

## 1.203 jade.guide/Buffer Extremes

Buffer Extremes  
.....

- Function: buffer-end &optional BUFFER  
Create and return a new position object pointing to the character after the last character in the buffer.
  - Function: goto-buffer-end  
Set the cursor to the character after the last character in the current buffer.
  - Function: buffer-start &optional BUFFER  
Create a new position pointing to the first character in the buffer. Currently this is always the position '#<pos 0 0>' and the
-

BUFFER argument is ignored.

- Function: goto-buffer-start  
Set the cursor position to the first character in the buffer.

## 1.204 jade.guide/Character Movement

Character Movement

.....

- Function: left-char &optional COUNT POS  
Alter and return POS (or a copy of the cursor pos) so that it points COUNT characters (default is one) to the left of its current position. If the resulting column number is less than zero 'nil' is returned, else the position.

```
(goto-char (pos 20 0))
=> #<pos 20 0>
(left-char)
=> #<pos 19 0>
```

```
(setq x (pos 4 1))
=> #<pos 4 1>
(left-char 3 x)
=> #<pos 1 1>
x
=> #<pos 1 1>
```

- Function: goto-left-char &optional COUNT  
Move COUNT (or one) characters to the left.
- Function: right-char &optional COUNT POS  
Alter and return POS (or a copy of the cursor pos) so that it points COUNT (or one) characters to the right of its current position. May return a position which points to a character past the end of the line.
- Function: goto-right-char &optional COUNT  
Move COUNT (or one) characters to the right.

The following functions results depends on the contents of the buffer they are operating on; they move a certain number of \*characters\*, and hence will cross line boundaries.

- Function: next-char &optional COUNT POS BUFFER  
Alter and return POS (or a copy of the cursor pos) to point to the character COUNT characters in front of its current position.

If COUNT is negative this function will work backwards through the buffer.

- Function: goto-next-char &optional COUNT  
Move COUNT characters forwards.

- Function: `prev-char &optional COUNT POS BUFFER`  
Similar to the `'next-char'` function but will work backwards when `COUNT` is positive and forwards when it is negative.
- Function: `goto-prev-char COUNT`  
Move `COUNT` characters backwards.

## 1.205 jade.guide/Word Movement

### Word Movement

.....

There are two buffer-local variables which control the syntax of words in each buffer.

- Variable: `word-regexp`  
This buffer-local variable contains a regular expression which will match each character allowed to be in a word.

The standard value is `'[a-zA-Z0-9]'`, i.e. all alphanumeric characters.

- Variable: `word-not-regexp`  
A buffer-local variable. Holds a regular expression which will match anything not in a word.

The normal value is `'[^a-zA-Z0-9]|$'`, i.e. anything which is not alphanumeric or the end of a line.

The following functions use these variables when deciding what is and what isn't a word.

- Function: `forward-word &optional COUNT POS MOVE`  
Return the position of the first character after the end of the word at position `POS` (or the cursor). `COUNT` is the number of words to move, negative values mean go backwards.  
  
If `MOVE` is non-`'nil'` then the cursor is moved to the result.  
  
Note that `POS` is not altered.
  - Function: `backward-word &optional COUNT POS MOVE`  
Similar to `'forward-word'` except that it works backwards. In fact, all this function does is call `'forward-word'` with `COUNT` negated.
  - Function: `word-start &optional POS`  
Returns the position of the first character of the word at `POS` (or the cursor position).
  - Function: `in-word-p &optional POS`  
This function returns `'t'` if `POS` (or the cursor) is in a word.
-

## 1.206 jade.guide/Tab Movement

### Tab Movement

.....

- Function: `prev-tab &optional COUNT POS SIZE`  
Alter and return POS (or a copy of the cursor position) so that it points COUNT (default is one) tab stops to the left of its current position. Returns 'nil' if that position is before the start of the line.

SIZE is optionally the number of glyphs in each tab, or the value of the 'tab-size' variable.

Note that the position returned is not the position of a character but of a glyph (see  
Glyph Positions  
).

```
(prev-tab 1 (pos 20 0))
=> #<pos 16 0>
```

- Function: `goto-prev-tab &optional COUNT SIZE`  
Move COUNT tab stops to the left.
- Function: `next-tab &optional COUNT POS SIZE`  
Alter and return POS (or a copy of the cursor position) so that it points COUNT tab stops to the right of its current position.  
  
SIZE is optionally the number of glyphs in each tab, or the value of the 'tab-size' variable.  
  
Note that the position returned is not the position of a character but of a glyph (see  
Glyph Positions  
).
- Function: `goto-next-tab &optional COUNT SIZE`  
Move COUNT tab stops to the right.

## 1.207 jade.guide/Line Movement

### Line Movement

.....

- Function: `next-line &optional COUNT POS`  
Alter and return POS (or a copy of the cursor position) so that it points COUNT (or one) lines forwards, the column component is not changed.

If COUNT is negative (i.e. go backwards) and the resulting line number is less than zero 'nil' is returned.

```
(next-line 2 (pos 1 1))
=> #<pos 1 3>
```

```
(next-line -1 (pos 1 1))
=> #<pos 1 0>
```

- Function: goto-next-line &optional COUNT  
Move COUNT lines downwards, the column number of the cursor is adjusted so that its glyph position is as close to its previous glyph position as possible.
- Function: prev-line &optional COUNT POS  
Similar to NEXT-LINE but goes backwards (or forwards with a negative COUNT).
- Function: goto-prev-line &optional COUNT  
Move COUNT lines backwards, adjusting the column number of the cursor as necessary.

## 1.208 jade.guide/Expression Movement

### Expression Movement

.....

Some major modes provide functions to move backwards and forwards over expressions written in a buffer in the programming language that the mode supports (see

Mode-Specific Expressions

), for example the Lisp

mode defines the syntax of Lisp forms written in a buffer.

- Function: forward-exp &optional COUNT  
This function moves the cursor over COUNT expressions, as defined in the current buffer. If the buffer has no expression definitions an error is signalled.
- Function: backward-exp &optional COUNT  
Moves backwards over COUNT (or one) expressions, leaving the cursor at the beginning of the expression. If the buffer has no expression definition functions an error is signalled.

## 1.209 jade.guide/Positions and Offsets

### Positions and Offsets

-----

Although Jade stores the position of a character as a pair of two numbers many other programs define the position of a character as its

offset from the beginning of the buffer or file it is in. The following functions may be used to convert between these two types of positions in a specified buffer.

- Function: `pos-to-offset` &optional POS BUFFER

This function returns the offset of the character at the position POS (or the cursor position by default) in the specified buffer. This will be an integer, the first character in a buffer is represented by an offset of zero.

```
(pos-to-offset (pos 0 0))
=> 0
```

```
(pos-to-offset)
=> 195654
```

- Function: `offset-to-pos` OFFSET &optional BUFFER

Creates a new position object which contains the position of the character OFFSET characters from the start of the specified buffer.

```
(offset-to-pos 0)
=> #<pos 0 0>
```

```
(offset-to-pos 195654)
=> #<pos 14 5974>
```

## 1.210 jade.guide/Marks

### Marks

=====

A "mark" is a Lisp object which points to a character in a file (or buffer), as the buffer the file is stored in is modified the position the mark points to is also modified so that the mark will *\*always\** point to the same character.

The character that a mark points to does not have to be loaded into the editor all the time either; if the file the character is in is not resident in a buffer the mark will simply contain the character's position and the file's name. When a file is loaded any marks pointing to the file are altered so that they point straight to the buffer containing the file.

- Function: `markp` OBJECT

This function returns 't' if its argument is a mark.

### Mark Components

Marks contain two values; position and file

### Mark Relocation

How the position of a mark is updated as its buffer is modified

#### Mark Residency

Marks may point to files which have not been loaded

#### Creating Marks

Functions to allocate new mark objects

#### Altering Marks

Setting the components of a mark

#### Moving to Marks

Moving the cursor to the character a mark points to

## 1.211 jade.guide/Mark Components

### Mark Components

-----

Each mark object has two main components; the position of the character pointed to by the mark (a position object) and the file which the character is contained by.

The file is the most complex component, it can be either a string naming the file or a buffer. When the file component is a string the mark is said to be "non-resident" since none of the editor buffers contain the character which the mark points to.

- Function: mark-pos MARK

Returns the position object contained in the marker MARK, no copy is made: if you modify the position returned it will be reflected in the position of the mark.

Note that if you later modify the buffer the mark is resident in the position previously returned by 'mark-pos' may be altered by the mark relocation process. See

Mark Relocation

.

- Function: mark-file MARK

Returns the file component of MARK. This will be either the name of the file or the buffer itself depending on whether the mark is resident or not. See

Mark Residency

.

## 1.212 jade.guide/Mark Relocation

---



## Mark Relocation

---

An important feature of marks is that they always point to the same character, even when the buffer has been modified, changing the position of the character (i.e. if some text is deleted from somewhere before the character its position will probably change).

Every time a buffer is modified each mark which points to a character in that buffer is examined and then, if necessary, the position it points to is changed to take account of the buffer's new state.

Basically, what happens is that each mark will try to point at the same character all the time. If some text is inserted at the position of the mark the mark's position will be advanced to the end of the insertion and hence the original character.

The only time the mark will not point at the same character is when the character is deleted from the buffer. In this case the mark will point to the start of the deletion.

### 1.213 jade.guide/Mark Residency

#### Mark Residency

---

As I have already explained, a mark does not necessarily have to point at a character loaded into a buffer; it can also point at a character in a file on disk somewhere. When this happens the mark is said to be non-resident.

- Function: `mark-resident-p MARK`

This function returns 't' when the character pointed to by the marker `MARK` is resident in one of the editor's buffers.

When the function 'mark-file' (see

Mark Components

) is applied to a

non-resident mark it returns the full name of the file, for example,

```
(setq x (make-mark (pos 0 20) "/tmp/foo.c"))
=> #<mark "/tmp/foo.c" #<pos 1 21>>
(mark-resident-p x)
=> nil
(mark-file x)
=> "/tmp/foo.c"
```

When a file is loaded into a buffer all existing non-resident marks are examined to see if they point to that file. If so that mark has its file component set to the buffer that the file was loaded into.

Similarly, when a buffer is deleted any marks pointing to characters in that buffer are made non-resident: their file component is set to the

---

name of the file.

When the function which moves the cursor to the position of a specific mark (`'goto-mark'`, see `Moving to Marks`) is called with a non-resident mark it will try to load the file into a buffer.

The following code fragment can be used to ensure that a mark MARK is resident,

```
(or (mark-resident-p MARK)
 (open-file (mark-file MARK))
 (error "Can't make mark resident, %S" MARK))
```

## 1.214 jade.guide/Creating Marks

### Creating Marks

---

- Function: `make-mark` &optional POS BUFFER-OR-FILENAME  
This function allocates a new mark object and fills it in according to the supplied arguments.

It will point at a character at position POS, or the position of the cursor in the current window. Note that a copy of POS is made.

The BUFFER-OR-FILENAME argument specifies the file component of the mark. If BUFFER-OR-FILENAME is a buffer (`'nil'` or undefined means the current buffer) the mark will use it and therefore will be resident (see `Mark Residency`).

Alternatively, BUFFER-OR-FILENAME can be a string naming the file explicitly. If the file is already loaded into a buffer that buffer will be used and the mark will be resident. Otherwise the mark will be non-resident and the string will be used as the file component.

With no arguments this function will produce a resident mark pointing at the cursor in the current buffer.

```
(make-mark)
=> #<mark #<buffer programmer.texi> #<pos 46 6152>>

(make-mark (buffer-start) "/tmp/foo")
=> #<mark "/tmp/foo" #<pos 0 0>>

(make-mark (pos 0 3))
=> #<mark #<buffer programmer.texi> #<pos 0 3>>
```

---

## 1.215 jade.guide/Altering Marks

### Altering Marks

---

If you just want to set the position of a mark you can modify its position component (see

Mark Components

). Alternately the following

function may be used. When you need to set the file a mark points to the only method is to use this function.

- Function: `set-mark MARK &optional POS BUFFER-OR-FILENAME`

This function sets either or both of the position and file components of the mark object MARK, then returns MARK.

If POS is a position object the position component of MARK will be set to it (a copy of it actually).

If the BUFFER-OR-FILENAME argument is non-`'nil'` the file component of MARK will be set. This argument can be a buffer object or a string naming a file. If a named file is already in a buffer that buffer will be used instead.

```
(setq x (make-mark))
=> #<mark #<buffer programmer.texi> #<pos 46 6186>>
(set-mark x (buffer-start))
=> #<mark #<buffer programmer.texi> #<pos 0 0>>
(set-mark x nil "/tmp/foo")
=> #<mark "/tmp/foo" #<pos 0 0>>
```

## 1.216 jade.guide/Moving to Marks

### Moving to Marks

---

- Function: `goto-mark MARK`

This function switches to the buffer containing MARK (if necessary) and then moves the cursor to the character that the mark points to.

If the mark is not currently resident an attempt will be made to load the mark's file into a new buffer and use that.

## 1.217 jade.guide/Glyph Tables

---

## Glyph Tables

=====

A "glyph table" is a Lisp object used to define a mapping between the characters which may occur in a buffer (anything with a numeric value between 0 and 255 inclusive) and the sequences of glyphs which are drawn into a window to represent these characters.

A "glyph" is a image which, when rendered into the display, takes up one character position. Each character in a buffer is rendered as a sequence of 1 or more glyphs.

- Function: `glyph-table-p` OBJECT  
This function returns `'t'` when its argument is a glyph table.

### Glyph Table Basics

How a glyph table defines mappings

### Glyph Positions

The position of a character and its glyph sequence may be different

### Creating Glyph Tables

Making new glyph tables

### Buffer Glyph Tables

Each buffer may use a separate glyph table for its display

## 1.218 jade.guide/Glyph Table Basics

### Glyph Table Basics

-----

A glyph table is basically an array that has 256 elements; each element represents one character and contains between zero and four glyphs -- the glyphs which will be printed for the character.

A special case exists for the tab character; when an element in the table contains zero glyphs, enough spaces will be printed to fill in to the next tab stop.

- Function: `get-glyph` GLYPH-TABLE CHARACTER  
This function returns a string containing the glyphs in the element of the glyph table GLYPH-TABLE for the character CHARACTER.

```
(get-glyph (default-glyph-table) ?a)
=> "a"
```

```
(get-glyph (default-glyph-table) ?\t)
```

```

=> "" ;TAB is special

(get-glyph (default-glyph-table) ?\000)
=> "^@" ;the NUL character

```

- Function: `set-glyph GLYPH-TABLE CHARACTER GLYPH-STRING`

This function sets the sequence of glyphs used to render the character `CHARACTER` in the glyph table `GLYPH-TABLE` to the characters in the string `GLYPH-STRING`.

An error is signalled if there are more than four characters in `GLYPH-STRING`.

All buffers which use `GLYPH-TABLE` for their rendering will be totally redrawn at the next redisplay.

## 1.219 jade.guide/Glyph Positions

### Glyph Positions

-----

Position objects are usually used to refer to the position of a character in a buffer, this position (sometimes called the "character position" may not be the same as the position of the sequence of glyphs printed to represent the character. When a position object is used to refer to the position of a glyph it is called a "glyph position".

For example, consider a line in a buffer containing the string `'a\tb'` (where `'\t'` represents a tab character). When this is rendered in a buffer the glyphs which will actually be drawn are,

```

a b

```

That is, an `'a'` glyph, followed by seven (assuming `'tab-size'` is set to 8) `' '` glyphs, and lastly a `'b'` glyph.

The character position of the `'b'` character in the buffer is `'#<pos 2 LINE>'`, where `LINE` is the line's number.

Now the confusing bit: the `*glyph*` position of the `'b'` `*glyph*` is `'#<pos 8 LINE>'` since it is actually the ninth glyph to be drawn.

The good news is that most of the time you can forget about glyph positions, they only need to be considered when you're thinking about how the buffer will look when rendered in the window. For example, Lisp programs which indent source code will definitely need to use glyph positions.

Two functions are provided for converting between character and glyph positions and vice versa.

- Function: `char-to-glyph-pos &optional POS BUFFER`

Return a new position object containing the glyph position of the character at character position `POS` (or the cursor position) in the

specified buffer.

- Function: `glyph-to-char-pos` POS &optional BUFFER  
This function returns a new position object containing the character position of the glyph printed at glyph position POS in the specified buffer.

If the glyph position POS is not the position of the first in a sequence of glyphs representing a single character the position of the next character will be returned.

## 1.220 jade.guide/Creating Glyph Tables

Creating Glyph Tables  
-----

- Function: `make-glyph-table` SOURCE  
This function creates a new glyph table, containing glyph sequences defined by the SOURCE argument.

If SOURCE is a glyph table it will be copied, if it's a buffer that buffer's glyph table will be copied or if SOURCE is 'nil' a copy of the default glyph table will be made.

## 1.221 jade.guide/Buffer Glyph Tables

Buffer Glyph Tables  
-----

Each buffer may define its own glyph table that will be used to provide the character-to-glyph mappings for that buffer.

- Function: `buffer-glyph-table` &optional BUFFER  
Returns the glyph table installed in the buffer.
- Function: `set-buffer-glyph-table` GLYPH-TABLE &optional BUFFER  
Sets the glyph table being used in the buffer to GLYPH-TABLE.

By default, each buffer uses the "default glyph table". This is a glyph table set up when the editor initialise itself. The mappings it provides are very generic, for more details see  
Character Images  
.

- Function: `default-glyph-table`  
This function returns the default glyph table.

Redefining some of the mappings in the default glyph table is an easy way to affect rendering operations, for example if I want the UK pound sign character (ASCII value is octal 243) to be printed as itself and

---

not the usual escape sequence I can do the following,

```
(set-glyph (default-glyph-table) ?243 "\243")
```

## 1.222 jade.guide/Input Events

### Input Events

=====

An "input event" is a Lisp object representing an action initiated by the user, i.e. a key press, pressing a mouse button and similar things.

Note that input events are often referred to as key presses, this isn't really accurate but since most input events are key presses the term sort of stuck. Anyway, wherever the phrase 'key press' occurs in this manual it could be replaced by 'input event'.

Each input event is represented by a cons cell (see  
 Cons Cells  
 )

containing two integers, these integers encode the actual input event. The encoding is opaque; the only way to access an event meaningfully is via the functions provided.

- Function: `eventp OBJECT`  
 This function returns `t` if its argument is an input event.

Each event has a textual name, for the actual format of these names see

Key Names

.

Functions are available to convert between the name of an event and the actual event itself, and vice versa.

- Function: `lookup-event EVENT-NAME`  
 Create and return a new input event whose name is `EVENT-NAME`.

```
(lookup-event "Ctrl-x")
=> (120 . 9)
```

```
(lookup-event "Ctrl-Meta-LMB-Click1")
=> (1 . 58)
```

- Function: `event-name EVENT`  
 This function returns a string naming the input event `EVENT`.

```
(event-name (lookup-event "Ctrl-x"))
=> "Ctrl-x"
```

## 1.223 jade.guide/Keymaps

### Keymaps

=====

A "keymap" is a Lisp object defining a mapping between input events (see

Input Events

) and commands to be executed when the event loop (see

Event Loop

) receives the input event.

- Function: `keymapp OBJECT`

Returns `t` when `OBJECT` is a keymap.

### Types of Keymap

Two different formats of keymap

### Creating Keymaps

Allocating new keymaps

### Binding Keys

Inserting and removing key bindings

### Key Lookup

How a key press is resolved into a command

### Prefix Keys

Chaining events into multiple-event bindings

### Standard Keymaps

Predefined keymaps you can modify

## 1.224 jade.guide/Types of Keymap

### Types of Keymap

-----

There are two different types of keymap; one for keymaps which contain only a few bindings, the other providing a more efficient method of storing larger numbers of bindings.

#### "Key lists"

These are used for keymaps which only contain a few bindings; they are lists whose first element is the symbol `'keymap'`. All subsequent elements define bindings, they are represented by three-element vectors. The first two are the contents of the cons cell representing the input event, the other element is the



command to be invoked.

For example,

```
(keymap [120 9 some-command])
```

Since the event `(120 . 9)` is the key press `Ctrl-x`, this keymap binds the command `some-command` to the key press `Ctrl-x`.

### "Key tables"

Key tables are used for keymaps which contain a larger number of bindings. They are vectors of 127 elements, a hash function is used to hash each event contained in the keymap into one of the 127 buckets. Each bucket is a list of key bindings in the same form as a key list (but without the `keymap` symbol).

## 1.225 jade.guide/Creating Keymaps

### Creating Keymaps

-----

Since there are two different types of keymap (lists and tables) there are two different functions for creating them with.

- Function: `make-keylist`

Creates and returns a new key list containing no bindings.

```
(make-keylist)
=> (keymap)
```

- Function: `make-keytab`

This function returns a new key table; it will be totally empty.

```
(make-keytab)
=> [nil nil ... nil]
```

If you want to produce a new copy of a keymap use the `copy-sequence` function (see

```
Sequence Functions
) to duplicate the source keymap.
```

## 1.226 jade.guide/Binding Keys

### Binding Keys

-----

The `bind-keys` function is used to install new key bindings into a keymap (either a key list or table).

---

- Function: `bind-keys KEYMAP &rest BINDINGS`

This function installs zero or more key bindings into the keymap `KEYMAP`.

Each binding is defined by two elements in the list of `BINDINGS`, the first defines the name of the input event (or the event itself) and the second defines the command to be associated with the event.

For example to bind two keys in the keymap `KEYMAP`; the event `'Ctrl-f'` to the command `'goto-next-char'` and the event `'Ctrl-b'` to the command `'goto-prev-command'` the following form would be used,

```
(bind-keys KEYMAP
 "Ctrl-f" 'goto-next-char
 "Ctrl-b" 'goto-prev-char)
```

- Function: `unbind-keys KEYMAP &rest KEYS`

This function removes the bindings of the events `KEYS` (these may be the names of the events or the event objects themselves) from the keymap `KEYMAP`.

```
(unbind-keys KEYMAP
 "Ctrl-f"
 "Ctrl-b")
```

## 1.227 jade.guide/Key Lookup

### Key Lookup

Each time the event loop (see `Event Loop`) receives an input event from the window system it searches for a binding of that event.

The variables `'keymap-path'` and `'next-keymap-path'` are used to determine the "keymap environment", this is the list of keymaps which are searched when looking for the binding.

- Function: `lookup-event-binding EVENT &optional RESET-PATH`  
This function examines the current keymap environment for a binding of the event `EVENT` (see `Input Events`). If such a binding is found its command is returned, otherwise `'nil'` is returned.

If the optional `RESET-PATH` argument is non-`'nil'` the `'next-keymap-path'` variable will be set to `'nil'`, otherwise it will be left with its original value.

- Variable: `keymap-path`  
A buffer-local variable providing the list of keymaps (or variables whose values are keymaps) which will be searched for a binding when the value of the `'next-keymap-path'` variable is `'nil'`.

```
keymap-path
=> (minor-mode-keymap texinfo-keymap global-keymap)
```

- Variable: next-keymap-path  
This variable is used to create multi-event key bindings. When it has a non-`'nil'` value it overrides the `'keymap-path'` variable when a key binding is being searched for.

After the value of this variable is used to search for a key binding it is set to `'nil'`. This means that, unless another prefix key occurred, the next input event received will be resolved through the `'keymap-path'` variable.

When this variable is set the value of the `'prefix-arg'` variable is set to the current value of the `'current-prefix-arg'` variable. This is so a prefix argument given to a multi-event command is transmitted through to the command.

For more details on multi-event bindings see  
Prefix Keys

.

## 1.228 jade.guide/Prefix Keys

Prefix Keys

-----

As briefly noted in the previous section it is possible to create multi-event key bindings. The `'next-keymap-path'` variable is used to link key presses (known as "prefix keys" since they prefix the actual, command-invoking, binding) to a new keymap environment which will be used to resolve the next key press. This method allows key sequences of an arbitrary length to be used.

The best way to explain this is probably with an example. Consider the following,

```
(setq entry-keymap (make-keylist))
(bind-keys entry-keymap
 "Ctrl-x" '(setq next-keymap-path '(second-keymap)))

(setq second-keymap (make-keylist))
(bind-keys second-keymap
 "Ctrl-j" 'some-command)
```

Two keymaps are created, the first of which, `'entry-keymap'`, would be placed in the `'keymap-path'` list. When `'Ctrl-x'` is typed the associated command would be invoked, installing the next piece of the chain, the `'second-keymap'` into the `'next-keymap-path'` variable.

So, after `'Ctrl-x'` is typed the keymap environment will be the list of keymaps `'(second-keymap)'`, subsequently typing `'Ctrl-j'` would then

invoke the command `'some-command'`.

## 1.229 jade.guide/Standard Keymaps

### Standard Keymaps

---

Several keymaps are predefined by Jade.

#### `'global-keymap'`

This keymap is the root of the global keymap structure; all buffers which allow themselves to be edited have this keymap in their `'keymap-path'`.

#### `'ctrl-x-keymap'`

This is linked to the `'global-keymap'` via the key `'Ctrl-x'`.

#### `'ctrl-x-4-keymap'`

The keymap for the global prefix `'Ctrl-x 4'`.

#### `'ctrl-x-5-keymap'`

The keymap for the global prefix `'Ctrl-x 5'`.

#### `'user-keymap'`

This keymap is only to be bound by the `*user*`, not by programmers! It's linked to the global prefix `'Ctrl-c'` and is intended to allow users to bind unmodified keys (modified keys with the prefix `'Ctrl-c'` are usually bound to by modes) to commands which don't have bindings by default.

## 1.230 jade.guide/Event Loop

### Event Loop

---

Whenever Jade is not executing a command it is sitting in the "event loop". This is where the editor waits for any input events which the window system sends it, invokes the commands they resolve to and then redraws all the editor windows to reflect the modifications made to any buffers.

#### Event Loop Actions

What actually happens

#### Commands

Commands are Lisp functions which may be called interactively by the user

---

|                 |                                               |
|-----------------|-----------------------------------------------|
| Event Loop Info | Information about the event loop              |
| Recursive Edits | How to call the event loop from Lisp programs |
| Reading Events  | Reading single events in Lisp                 |
| Idle Actions    | What happens when nothing happens             |

## 1.231 jade.guide/Event Loop Actions

### Event Loop Actions

-----

When Jade appears to be doing nothing it is probably sitting in the event loop waiting for input to arrive. When an input event arrives from the window system it is processed according to its type.

If the input event is a keyboard or mouse button event it is converted into a Lisp input event (see [Input Events](#)) and the current keymap environment is searched for a binding of that event (see

[Key Lookup](#)). If a binding of the event is found it defines a command (see [Commands](#)) to be invoked, the 'call-command' function (see [Calling Commands](#)) is used to do this.

When no binding of a key or mouse button event exists the hook, 'unbound-key-hook', is evaluated; if this returns 'nil' and the event is a keyboard event and no prefix keys (see [Prefix Keys](#)) preceded it the key is inserted into the current buffer before the cursor.

If the event was not a keyboard or mouse button event the event loop will deal with it itself; these events are generally things which should be transparent to Lisp programs (i.e. window exposure notification, etc...).

One exception is the event sent when a window should be closed (i.e. hitting the close-window gadget in Intuition, or sending a window the delete-window atom in X), the hook 'window-closed-hook' is called. By default this hook is setup to invoke the 'close-window' command (as

---

bound to `Ctrl-x 0`).

Another function of the event loop is to wait for input from any of the subprocesses currently executing (see

Processes

); whenever input is

pending in a subprocess's standard output channel it is copied to the process objects's output stream.

After processing an event or piece of subprocess output the event loop will redisplay any part of any window which needs to be updated; this may be necessary if a window is now displaying a different part of a buffer, or if the part of the buffer it is displaying has been modified. See

Rendering

.

Normally Jade will `sleep` while it's waiting for input, however after every second it spends asleep the event loop will wake up and try to do a sequence of operations; for more details see

Idle Actions

.

- Hook: unbound-key-hook

The hook called when an unbound input event is received.

- Hook: window-closed-hook

The hook called when an event is received telling Jade to close a window; the current window is the one which should be closed.

## 1.232 jade.guide/Commands

Commands

-----

A "command" is a Lisp function which may be called interactively, that is, either as a binding of an input event or by name (with the `Meta-x` key sequence).

Commands are defined in the same way as functions, using the `defun` special form; the body forms of a command must contain an "interactive declaration". This shows that the function may be called interactively part and tells the `call-command` function how to compute the argument values to apply to the command.

Interactive Declarations

How to define a command

Prefix Arguments

Arguments to a command from the user

### Calling Commands

The function used to invoke a command

### Example Commands

A definition of a command

## 1.233 jade.guide/Interactive Declarations

### Interactive Declarations

.....

When you define a command (using the `'defun'` special form in the same way you would define a function) the first of its body forms (after the optional documentation string) *must* be an interactive declaration.

This form looks like a call to the special form `'interactive'`, in actual fact this special form always returns `'nil'` and has no side-effects. The only effect of this form is to show the `'call-command'` function, which invokes commands, that this function definition is actually a command (i.e. it may be called interactively). The second element of the declaration form (after the `'interactive'` symbol) defines how the argument values applied to the command are computed.

The structure of an interactive declaration, then, is:

```
(interactive [CALLING-SPEC])
```

When a command is defined this is how it is defined with the interactive declaration:

```
(defun some-command (arg1)
 "Optional documentation string."
 (interactive ...)
 ...)
```

The `CALLING-SPEC` form defines the argument values applied to the command when it is called interactively, it may be one of,

- \* `'nil'` or undefined (i.e. `'(interactive)'`); no arguments are given to the command, this type of interactive declaration just shows that the function may be called interactively.
- \* A string; zero or more lines (each separated by a newline character), each line defines how to compute one argument value. The first character of each line is a code letter defining exactly how to compute the argument, the rest of the line is an optional prompt string which some code letters show the user when prompting for the argument.

The currently available code letters are,

`'a'`

---

- Prompt, with completion, for a function object.
- ``b'`  
Prompt, with completion, for an existing buffer object.
- ``B'`  
Prompt, with completion, for a buffer; if it doesn't yet exist it will be created.
- ``c'`  
Prompt for a character.
- ``C'`  
Prompt with completion for a command.
- ``d'`  
The position of the cursor in the current window.
- ``D'`  
Prompt with completion for the name of a directory in the filing system.
- ``e'`  
The event which caused this command to be invoked.
- ``E'`  
The event which caused this command, cooked into a string.
- ``f'`  
Prompt with completion for the name of an existing file.
- ``F'`  
Prompt with completion for the name of a file; it doesn't have to exist.
- ``k'`  
Prompt for a single event.
- ``m'`  
The starting position of the marked block in the current window.
- ``M'`  
The ending position of the current block.
- ``n'`  
Prompt for a number.
- ``N'`  
The prefix argument (see  
Prefix Arguments  
) as a number, if no  
prefix argument exists, prompt for a number.
- ``p'`  
The prefix argument as a number, this will be 1 if no prefix  
argument has been entered.
-



`'p'`  
The raw prefix argument.

`'s'`  
Prompt for a string.

`'S'`  
Prompt with completion for a symbol.

`'t'`  
The symbol `'t'`.

`'v'`  
Prompt with completion for a variable.

`'x'`  
Read one Lisp object.

`'X'`  
Read a Lisp object, then evaluate it.

A null line produces an argument value of `'nil'`.

Any non-alphabetic characters at the beginning of the CALLING-SPEC are used as flags, the currently recognised flags are,

`'*''`  
If the active buffer is read-only an error will be signalled.

`'_'`  
After building the argument list the block marked in the current window will be unmarked.

\* Anything else; the form is evaluated and expected to return a \*list\* of arguments to apply to the command.

Some example interactive declarations,

```
;; No arguments, but the function may be called
;; as a command.
(interactive)

;; One argument, an existing buffer
(interactive "bBuffer to kill:")

;; If buffer isn't read-only, three arguments:
;; 'nil', a Lisp object and 't'.
(interactive "*\nLisp form:\nt")
```

## 1.234 jade.guide/Prefix Arguments

---

## Prefix Arguments

.....

When the you invoke a command it is often useful to be able to specify arguments which the command will act on. "Prefix arguments" are used for this purpose. They are called `*prefix*` arguments since they are entered before the command is invoked, and therefore prefix the command with an argument. Prefix arguments are usually integers.

The easiest way for a command to access these arguments is through its interactive declaration (see `Interactive Declarations`) and the `'N'`, `'p'` and `'P'` code letters.

The two variables `'prefix-arg'` and `'current-prefix-arg'` are used to store prefix arguments. Whenever a command is invoked the value of `'prefix-arg'` is moved to `'current-prefix-arg'` and `'prefix-arg'` set to `'nil'`. This allows commands to set the prefix argument of the next command by assigning a value to the `'prefix-arg'` variable.

These variables store an object known as the "raw prefix argument", when a command is called it normally uses the "numeric prefix argument", this is an integer created from the raw argument using the following rules,

- \* If the raw arg is `'nil'` the numeric value is 1.
- \* If the raw arg is any other symbol the value is -1.
- \* A number is used unchanged.
- \* A cons cell stores the numeric value in its car.

The `'prefix-numeric-argument'` function is used to convert the raw argument into a numeric value.

- Function: `prefix-numeric-argument RAW-ARG`  
Returns the numeric value of the raw prefix argument `RAW-ARG`.
- Variable: `prefix-arg`  
The value of the raw prefix argument used by the next command to be invoked.
- Variable: `current-prefix-arg`  
The value of the raw prefix argument of the current command.

## 1.235 jade.guide/Calling Commands

### Calling Commands

.....

When a command is to be invoked, the `'call-command'` function is

---

used. This builds a list of argument values to apply to the command (using its interactive declaration) then calls the command.

- Function: `commandp OBJECT`  
 This function returns `t` if its argument may be called interactively. If `OBJECT` is a function (i.e. a symbol or a lambda-expression) it is a command if it contains an interactive declaration (see `Interactive Declarations`).

The only other object which is a command is a function call form; the use of these types of commands is discouraged but they can be useful sometimes.

```
(commandp 'setq)
=> nil
```

```
(commandp 'isearch-forward)
=> t
```

```
(commandp '(setq x 20))
=> t
```

- Command: `call-command COMMAND &optional PREFIX-ARG`  
 This function calls the command `COMMAND` interactively. See the documentation of `'commandp'` above for what constitutes a command.

If the `PREFIX-ARGUMENT` is non-nil it defines the value of the `'current-prefix-arg'` variable for this command, normally the value of this variable would be taken from the global `'prefix-arg'` variable.

When called interactively, this function will prompt for a command to invoke. This function is bound to the key sequence `'Meta-x'`.

## 1.236 jade.guide/Example Commands

Example Commands

.....

This is a couple of simple commands, taken from the source code of Jade.

```
(defun backward-kill-word (count)
 "Kill COUNT words backwards."
 (interactive "p")
 (kill-area (forward-word (- count)) (cursor-pos)))
```

```
(defun find-file (name)
 "Sets the current buffer to that containing the file NAME, if
NAME is unspecified it will be prompted for. If the file is not
already in memory 'open-file' will be used to load it."
```

```
(interactive "Find file: ")
(goto-buffer (open-file name))
```

## 1.237 jade.guide/Event Loop Info

### Event Loop Information

-----

- Variable: `this-command`  
This variable contains the value of the command currently being executed.
- Variable: `last-command`  
Holds the previously executed command.
- Function: `current-event`  
Returns the event which caused this command to be invoked.
- Function: `current-event-string`  
Returns a string which is the 'cooked' representation of the current event.
- Function: `last-event`  
Returns the event which caused the previous command.

## 1.238 jade.guide/Recursive Edits

### Recursive Edits

-----

Entering a "recursive edit" basically means to recursively call the event loop from a Lisp program, this latest instance of the event loop will work like the normal event loop (the "top level" event loop) until it is exited, at which point the Lisp program will regain control.

Recursive edits should be used sparingly since they can be very confusing for the user; they are mainly used to implement interactive user interfaces in the middle of a Lisp program or command. This can be achieved by installing a special set of key bindings for the duration of the recursive edit.

When programming with recursive edits \*a lot\* of care should be used; if proper cautions aren't taken an abnormal exit from a recursive error can wreak havoc.

Note that 'throw' and 'catch' (see  
Catch and Throw  
) can be used

\*through\* recursive edits with no problems; the recursive edit will automatically be aborted.

---

- Command: recursive-edit  
Enter a new level of recursive editing.
- Function: recursion-depth  
This function returns the number of recursive edits currently in progress. When in the top level this will return zero.
- Command: top-level  
Abort all recursive edits, control will be passed straight back to the top level event loop.
- Command: abort-recursive-edit &optional EDIT-VALUE  
This function aborts the outermost recursive edit (but *\*never\** the top level) returning EDIT-VALUE (or 'nil') from the instance of the 'recursive-edit' function which invoked this recursive edit.

When using recursive edits it is important to remember that the buffer and window configuration that existed when the edit was entered may not still exist when the recursive edit terminates. This means that some care has to be taken when installing and removing buffer-local values of variables. For example, the 'ask-y-or-n' function, which uses a recursive edit, does something like this:

```
(let
 ;; First save the old values of the variables to be altered.
 ;; The variables can't be directly bound to since this doesn't
 ;; work properly with buffer-local variables :- (
 (old-u-k-h unbound-key-hook)
 (old-k-p keymap-path)
 (old-buf (current-buffer)))
 ;; Now install the new values
 (setq unbound-key-hook (cons #'(lambda ()
 (beep)
 t)
 nil)
 keymap-path '(y-or-n-keymap)
 status-line-cursor t)
 ;; This is the important bit; ensure that the old values will
 ;; be reinstated even if an abnormal exit occurs. Also note
 ;; that they are always set in the original buffer.
 (unwind-protect
 (catch 'ask
 (recursive-edit))
 (with-buffer old-buf
 (setq keymap-path old-k-p
 unbound-key-hook old-u-k-h
 status-line-cursor nil))))))
```

## 1.239 jade.guide/Reading Events

Reading Events

-----

---

Most of the time it is unnecessary to read events manually; usually a special-purpose keymap will be sufficient. However it is possible to read single events from a Lisp program.

- Function: `read-event` &optional PROMPT-STRING  
Read the next input event from the current window and return it. If the optional string PROMPT-STRING is defined it is a one-line message to display while waiting for the event.

Note that this function isn't very efficient when used heavily; it uses a recursive edit and the `'unbound-key-hook'` to read the event. If possible use a keymap instead.

## 1.240 jade.guide/Idle Actions

### Idle Actions

-----

When a second goes by with no input events arriving, the editor assumes that it has "idle time" available, and tries to use this period to do non-essential tasks. These tasks include things like garbage collection and auto-saving modified files.

Whenever idle time is detected one of the following tasks is performed. They are listed in order of preference; once one of these has been done Jade will again sleep until an input event is received or another second elapses, whichever happens soonest.

1. If prefix keys have been entered and are outstanding their names will be printed in the status line. See  
    Prefix Keys  
    .
2. If any buffers are ready to be auto-saved (i.e. enough time since their last auto-save has elapsed) one of these buffers will be auto-saved. Only one buffer is ever saved in each idle period. See  
    Auto-Saving Files  
    .
3. If the total size of the data objects allocated since the last garbage collection is greater than the value of the `'idle-gc-threshold'` variable then the garbage collector is invoked.

- Variable: `idle-garbage-threshold`  
The number of bytes of Lisp data which must have been allocated since the last garbage collection for the garbage collector to be called in an idle period.

It is a good idea to set this variable much lower than the value of the `'gc-threshold'` variable since garbage collections happening while Jade is idle should usually be unnoticeable.

---

See

Garbage Collection

.

4. If none of the other tasks have been performed the 'idle-hook' hook is dispatched. I'm not sure what this hook could be used for but you never know...

## 1.241 jade.guide/Editing Files

Editing Files

=====

The main function of Jade is editing files of text; buffers (see

Buffers

) are used to contain files to be edited. When the buffer is displayed in a window (see

Windows

) the user can edit the file interactively using the keyboard and mouse.

This chapter documents the Lisp interface to all this; for the user's perspective see

Loading and Saving Files

.

Reading Files Into Buffers

How to read a file into a buffer

Writing Buffers

Functions to write buffers to files

Buffer Date Stamps

The last-modification time of each file is recorded

Buffer Modification Counts

Variables storing modification counts

Making Backups

How backup files can be made

Controlling Auto-Saves

Functions to control the auto-saving feature

## 1.242 jade.guide/Reading Files Into Buffers

### Reading Files Info Buffers

-----

Before a file can be edited it must be read into a buffer, this buffer can then be modified and later saved over the original contents of the file. Note that editing a buffer makes \*no\* changes to the contents of the file on disk; the buffer will have to be written back to the file on the disk first. See

Writing Buffers

.

- Function: open-file FILE-NAME

This function returns a buffer containing the contents of the file called FILE-NAME.

If an existing buffer contains the file called FILE-NAME that buffer is returned. Otherwise a new buffer is created and the file read into it.

When the file has successfully been read into the new buffer any local variables defined at the end of the file are processed (see

File Variables

) and the function 'init-mode' is used to try to install a major mode for the new buffer. See

Installing Modes

.

If file may not be written to the buffer is marked to be read-only.

Note that the hook, 'read-file-hook', can be used to read the contents of the file into the buffer if necessary. See the documentation of this hook for more details.

- Hook: read-file-hook

This hook is called by the 'open-file' function when it wants to read a file into a buffer. If the hook returns a non-'nil' value 'open-file' assumes that one member of the hook was successful in reading the file, otherwise the file will be read verbatim into the buffer.

The hook is called with two arguments: the name of the file and the buffer to read it into respectively.

If any members of the hook decide to read the file they're responsible for setting the 'buffer-file-name' component of the buffer and the buffer's 'buffer-file-modtime' variables to suitable values.

See the 'gzip.jl' file in the Lisp library directory for an example of how this hook can be used (in this case to automatically decompress gzip'ed files).

- Function: read-buffer FILE-OR-NAME &optional BUFFER

---



Replaces all text contained by the buffer by the contents of the file FILE-OR-NAME. This can be either a Lisp file object, in which case bytes will be read until the end of the file is reached, or the name of a file to read.

The following commands are used to read a file into a buffer then display that buffer in the current buffer.

- Command: `find-file FILE-NAME`

Display a buffer containing the file FILE-NAME in the current window.

When called interactively FILE-NAME will be prompted for.

- Command: `find-alternate-file FILE-NAME`

Replace the current buffer with one displaying the file FILE-NAME. What actually happens is that the current buffer is killed and a new one created.

When called interactively this function will prompt for its argument.

- Command: `find-file-read-only FILE-NAME`

Display a buffer containing FILE-NAME in the current window. The buffer will be read-only.

This will prompt for its argument when called interactively.

There is also a command to insert the contents of a file into a buffer.

- Command: `insert-file FILE-NAME &optional BUFFER`

This command inserts the contents of the file FILE-NAME into the buffer BUFFER (or the current buffer).

The hook 'insert-file-hook' is called with FILE-NAME as an argument to try and insert the file (into the current buffer at the current position). If this hook returns 'nil' (i.e. none of the functions in the hook inserted the file) it will be inserted normally.

If called interactively, FILE-NAME will be prompted for.

- Hook: `insert-file-hook`

Hook used to insert a file (given as the hook's argument) into the current buffer at the current cursor position.

- Command: `revert-buffer &optional BUFFER`

Reloads the contents of the buffer from the file it was originally loaded from; if any unsaved modifications will be lost the user is asked for confirmation.

---

## 1.243 jade.guide/Writing Buffers

### Writing Buffers

-----

After a buffer containing a file has been edited it must be written back to a file on disk, otherwise the modifications will disappear when Jade is exited!

- Function: `write-buffer &optional FILE-NAME BUFFER`  
The primitive to save a buffer's contents. The contents of the buffer `BUFFER` (or the current buffer) is written to the file `FILE-NAME` (or the `'buffer-file-name'` component of the buffer).
- Function: `write-buffer-area START-POS END-POS FILE-NAME &optional BUFFER`  
Writes the region of text from `START-POS` up to, but not including, `END-POS` to the file `FILE-NAME`.
- Function: `write-file BUFFER &optional FILE-NAME`  
Writes the contents of the buffer `BUFFER` to a file on disk. If the optional argument `FILE-NAME` is defined it names the file to write to. Otherwise, the value of the buffer's `'buffer-file-name'` component is used.

The hook `'write-file-hook'` is used to try and write the file, if this fails (i.e. the hook returns `'nil'`) the buffer is saved normally.

A backup may be made of the file to be overwritten (see

#### Making Backups

) and the protection-modes of the overwritten file will be preserved if possible.

- Hook: `write-file-hook`  
This hook is called by the `'write-file'` function when a buffer is to be saved. If no member of the hook actually writes the buffer to a file (i.e. the hook returns `'nil'`) `'write-file'` will do it itself in a standard way.

The hook function is responsible for creating any required backup file (use the function `'backup-file'`, see

#### Making Backups

) and

resetting the protection-modes of the new file to their original value.

See the file `'gzip.jl'` in the Lisp library directory for an example, it uses it to compress certain files automatically.

Remember to make sure that if a member of the hook writes the buffer it returns a non-`'nil'` value!

The following code fragment defines a function which does what the default action of `'write-file'` is,

```
(defun write-file-default-action (buffer name)
 (let
 ((modes (when (file-exists-p name) (file-modes name))))
 (backup-file name)
 (when (write-buffer name buffer)
 (when modes
 (set-file-modes name modes)
 t)))
```

The following commands call the 'write-file' function to write out a buffer, they also update the various variables containing information about the state of the buffer. It is normally unnecessary to call 'write-file' yourself; these commands should suffice.

- Command: save-file &optional BUFFER  
This command writes the buffer to the file that it was loaded from and then updates all the necessary buffer-local variables.

If the file on disk has been modified since it was read into the buffer the user is asked if they really want to save it (and risk losing a version of the file).

If no modifications have been made to the file since it was last saved it won't be saved again.

Any auto-saved version of the file is deleted.

- Command: save-file-as NEW-NAME &optional BUFFER  
This command saves the buffer BUFFER (or the current buffer) to the file called NEW-NAME. The 'buffer-file-name' is set to NEW-NAME and all the necessary buffer-local variables are updated.

If an auto-saved version of FILE-NAME exists it is deleted.

When called interactively NEW-NAME will be prompted for.

- Command: save-some-buffers  
For each buffer which contains unsaved modifications the user is asked whether or not to save the buffer.

't' is returned if no unsaved modifications exist in any buffers (i.e. the user replied 'yes' to all files which could be saved).

- Command: save-and-quit  
Calls 'save-some-buffers' then quits Jade (after asking the user if any unsaved buffers may be discarded).

## 1.244 jade.guide/Buffer Date Stamps

Buffer Date Stamps

-----

---

When a file is read into a buffer its (the file's) time of last modification is recorded, this can later be used to see if the file (on disk) has been modified since it was loaded into a buffer.

- Variable: `buffer-file-modtime`  
This buffer-local variable contains the file-modtime of the file stored in the buffer when it (the file) was last read from disk.

See

File Information

.

## 1.245 jade.guide/Buffer Modification Counts

Buffer Modification Counts

-----

Two buffer-local variables are used to record the modification count (see

Buffer Attributes  
) of a buffer when it is saved.

- Variable: `last-save-changes`  
A buffer-local variable containing the number of modifications made to the buffer the last time it was saved (either auto-saved or by the user).
- Variable: `last-user-save-changes`  
This buffer-local variable holds the number of modifications made to the buffer when it was last saved by the user.
- Variable: `last-save-time`  
A buffer-local variable holding the system time (from the `'current-time'` function) from when the buffer was last saved (auto-saved or by the user).

## 1.246 jade.guide/Making Backups

Making Backups

-----

For details of the variables which control whether and how backup files are made see

Backup Files

.

- Function: `backup-file FILE-NAME`  
When necessary, make a backup of the file `FILE-NAME`. This should be

called when the file FILE-NAME is about to be overwritten.

Note that this function doesn't define whether or not the file FILE-NAME will still exist when this function returns. Sometimes it will, sometimes it won't...

## 1.247 jade.guide/Controlling Auto-Saves

### Controlling Auto-Saves

---

For the documentation of the variables controlling the making of auto-save files see

Auto-Saving Files

.

- Function: `make-auto-save-name FILE-NAME`  
Returns a string naming the file which should hold the auto-saved version of the file FILE-NAME.

```
(make-auto-save-name "/tmp/foo")
=> "/tmp/#foo#"
```

- Function: `auto-save-function BUFFER`  
This function is called automatically whenever a buffer (BUFFER) needs to be auto-saved.

It firstly tries to use the 'auto-save-hook' hook to auto-save the file, if this fails (i.e. the hook returns 'nil') it is done manually (using the 'write-buffer' function).

- Hook: `auto-save-hook`  
Called by 'auto-save-function' (with the buffer as an argument) when a buffer is to be auto-saved.
  - Command: `delete-auto-save-file &optional BUFFER`  
This command deletes the auto-saved version of the buffer, if one exists.
  - Function: `auto-save-file-newer-p FILE-NAME`  
This function returns 't' when there is an auto-saved version of the file called FILE-NAME which is newer than FILE-NAME.
  - Command: `recover-file &optional BUFFER`  
If an auto-saved version of the buffer exists it is read into the buffer, overwriting its current contents. If any changes to the buffer will be lost the user is asked for confirmation.
-

## 1.248 jade.guide/Text

Text

====

This chapter describes all the functions used for editing and referencing the text stored in a buffer.

Note that where a command has a COUNT argument specifying the number of items to process; this argument will normally use the numeric value of the prefix argument when the function is called interactively.

Buffer Contents

    Accessing the contents of a buffer

Insertion Functions

    Inserting strings into a buffer

Deletion Functions

    Deleting regions of text

Kill Functions

    Recording regions of text

Transpose Functions

    Swapping two regions of text

Indentation Functions

    Functions for managing indentation

Translation Functions

    Applying a mapping to characters in a buffer

Search and Match Functions

    Regexps and general string matching

Rectangular Editing

    Manipulating rectangular regions

Controlling Undo

    How undo works

Misc Text Functions

    Other stuff

## 1.249 jade.guide/Buffer Contents

Buffer Contents

-----

- Function: `get-char` &optional POS BUFFER  
Returns the character at position POS (or the cursor position) in the specified buffer.
- Function: `set-char` CHARACTER &optional POS BUFFER  
Sets the character at position POS (or the cursor) in the buffer BUFFER (or the current buffer) to the character CHARACTER, then returns CHARACTER.
- Function: `copy-area` START-POS END-POS &optional BUFFER  
This function creates and returns a string containing the contents of the buffer BUFFER (or the current buffer) between the two positions START-POS (inclusive) and END-POS (exclusive).
- Function: `copy-block`  
If a block is marked in the current window returns a string containing the text marked then unmark the block, otherwise returns 'nil'.

If the marked block is rectangular the 'copy-rect' function (see

Rectangular Editing  
is used to get the string.

- Function: `clear-buffer` &optional BUFFER  
Removes all text from the specified buffer. No precautions are taken against losing any unsaved modifications that the buffer might contain!

## 1.250 jade.guide/Insertion Functions

### Insertion Functions

---

Note that the 'format' function can be used to provide formatted insertion; simply give it a suitable output stream. See  
Streams

.

- Command: `insert` STRING &optional POS BUFFER  
Inserts the string STRING into the specified buffer at the cursor position (or POS, if defined).  
  
Returns the position of the first character after the end of the inserted text.  
  
When called interactively the string to insert is prompted for.
  - Command: `insert-block` &optional POS  
If a block is marked in the current window, the text it contains is inserted at the position POS (or the cursor) and the block is unmarked.
-

If the marked block is rectangular the block is copied and inserted as a rectangle.

- Command: `yank &optional DONT-YANK-BLOCK`  
 Inserts a string before the cursor. If a block is marked in the current buffer and `DONT-YANK-BLOCK` is `'nil'` insert the text in the block. Else yank the last killed text. See  
     Kill Functions

.

When called interactively the raw prefix arg is used as the value of the `DONT-YANK-BLOCK` argument.

- Command: `yank-to-mouse`  
 Moves the cursor to the current position of the mouse pointer then calls the `'yank'` function.
- Command: `open-line COUNT`  
 Break the current line at the cursor, creating `COUNT` new lines. The cursor is left in its original position.
- Command: `split-line`  
 This function inserts a newline character (`'\n'`) at the current cursor position.

## 1.251 jade.guide/Deletion Functions

### Deletion Functions

-----

- Function: `delete-area START-POS END-POS &optional BUFFER`  
 This function deletes all text starting from the position `START-POS` up to, but not including, the position `END-POS`.

If `BUFFER` is defined it specifies the buffer to delete from, usually the current buffer is used.

- Function: `cut-area START-POS END-POS &optional BUFFER`  
 This function is a combination of the `'copy-area'` and `'delete-area'` functions; it copies the specified region then deletes it before returning the copy it made.

```
(cut-area START END)
==
(let
 ((text (copy-area START END)))
 (delete-area START END)
 text)
```

- Command: `delete-block`  
 Deletes the block marked in the current window (if one exists). This function knows about rectangular blocks.



- Function: cut-block  
Copies the block marked in the current window if one exists, then deletes it before returning the copied string. If the block is rectangular it is copied and cut as a rectangle.
- Command: delete-char COUNT  
Deletes COUNT characters, starting at the cursor position and working forwards.
- Command: backspace-char COUNT  
Deletes the COUNT characters preceding the cursor, if the cursor is past the end of the line, simply move COUNT characters to the left.

## 1.252 jade.guide/Kill Functions

### Kill Functions

-----

"Killing" a piece of text means to delete it then store a copy of it in a special place. This string is later available to other functions, such as 'yank' which inserts it into a buffer.

- Function: kill-string STRING  
This function adds the string STRING to the kill buffer. If the last command also killed something STRING is appended to the current value of the kill buffer.  
  
The 'this-command' variable is set to the value 'kill' to flag that the current command did some killing.  
  
Returns STRING.
  - Function: killed-string &optional DEPTH  
Returns the string in the kill buffer number DEPTH, currently only the last kill is stored so DEPTH must either be zero or undefined.
  - Command: kill-area START-POS END-POS  
This command kills a region of text in the current buffer, from START-POS up to, but not including, END-POS.  
  
When called interactively the currently marked block (if one exists) is used to provide the two arguments, then the block is unmarked.
  - Command: copy-area-as-kill START-POS END-POS  
Similar to 'kill-area' except that the region killed is not actually deleted from the buffer.
  - Command: kill-block  
Kills the block marked in the current window.
  - Command: copy-block-as-kill  
Kills the block marked in this window but doesn't actually delete
-

it from the buffer.

- Command: `kill-line` &optional ARG  
 This command kills lines from the cursor position. ARG is a raw prefix argument (see `Prefix Arguments`). What gets killed depends on ARG,
  - \* When ARG is 'nil' it kills from the cursor position to the end of the line, if the cursor is already at the end of the line it kills the newline character.
  - \* If the numeric value of ARG is greater than zero it kills from the cursor for that many whole lines.
  - \* If the numeric value is less than or equal to zero it kills that number of whole lines \*backwards\* from the cursor.
- Command: `kill-whole-line` COUNT  
 Kills \*all\* of the COUNT (an integer) next following lines.
- Command: `kill-word` COUNT  
 Kills COUNT words, starting at the cursor position.  
 When called interactively COUNT is the numeric prefix arg.
- Command: `backwards-kill-word` COUNT  
 Kills the COUNT previous words, starting from the cursor.  
 When called interactively COUNT is the numeric prefix arg.
- Command: `kill-exp` &optional COUNT  
 Kill COUNT expressions from the cursor position. See `Mode-Specific Expressions`.
- Command: `backward-kill-exp` &optional COUNT  
 Kills COUNT expressions, working backwards from the cursor. See `Mode-Specific Expressions`.

## 1.253 jade.guide/Transpose Functions

### Transpose Functions

---

"Transposing" two regions of text in a buffer means to swap their positions.

- Function: `transpose-items` FORWARD-ITEM-FUN BACKWARD-ITEM-FUN COUNT
-

This function transposes the areas defined by the functions FORWARD-ITEM-FUN and BACKWARD-ITEM-FUN (these functions must work in the style of 'forward-word' and 'backward-word' respectively).

What actually happens is that the item before the cursor is dragged forward over the next COUNT items.

- Command: transpose-words COUNT  
Uses 'transpose-items' with each item being a word.

When called interactively, COUNT is the value of the numeric prefix argument.

- Command: transpose-chars COUNT  
Transposes characters.
- Command: transpose-exps COUNT  
If the major mode in the current buffer has installed functions which define expressions then this command transposes expressions.  
See  
    Mode-Specific Expressions  
    .

## 1.254 jade.guide/Indentation Functions

### Indentation Functions

---

- Function: indent-pos &optional POS BUFFER  
This function returns the \*glyph\* position (see  
    Glyph Positions  
    )  
of the first character in the line pointed to by POS (or the cursor) which is not a TAB or SPC character.
  - Function: set-indent-pos INDENT-POS &optional BUFFER ONLY-SPACES  
Sets the indentation of the line pointed to by POS to the column pointed to by POS by putting the optimal sequence of TAB and SPC characters at the start of the line.  
  
If the ONLY-SPACES argument is non-'nil' no TAB characters will be used.
  - Command: indent-to COLUMN &optional ONLY-SPACES  
This function inserts enough TAB and SPC characters to move the cursor to glyph column COLUMN.  
  
If the ONLY-SPACES argument is non-'nil' no TAB characters are used.  
  
Note that COLUMN counts from zero.  
  
When called interactively the COLUMN argument is either the
-

numeric value of the prefix argument or, if no prefix argument has been entered, the result of prompting for a number.

- Command: `tab-with-spaces`

This command inserts enough spaces at the cursor position to move the cursor to the next tab stop.

Some major modes provide their own method of indentation (for example Lisp mode will indent Lisp programs in the proper style), see

Mode-Specific Indentation

.

- Command: `indent-line`

If the current buffer has a method for indentation installed, use it to indent the current line to its correct depth.

- Command: `newline-and-indent`

Insert a newline character, then indent the new line; if no function for indenting lines has been installed in this buffer a single TAB character is inserted.

- Command: `indent-area START-POS END-POS`

Uses the buffer's indentation method to indent all lines in the specified region to their correct depth.

When called interactively the currently-marked block is used to get the values of the two arguments, the block is then unmarked.

## 1.255 jade.guide/Translation Functions

### Translation Functions

-----

- Function: `translate-area START-POS END-POS TRANSLATION-TABLE &optional BUFFER`

This function applies the mapping `TRANSLATION-TABLE` to each character in the region starting at the position `START-POS` up to, but not including, `END-POS`.

`TRANSLATION-TABLE` is a string, each character represents the mapping for an ASCII character of that character's position in the string. If the string is less than 256 characters in length any undefined characters will remain unchanged (i.e. a `TRANSLATION-TABLE` of `' '` would leave the region unaltered).

- Function: `translate-string STRING TRANSLATION-TABLE`

This function uses a similar method to that used in the `'translate-area'` function. Instead of applying the mapping to a region of a buffer it applies it to the string `STRING`. `STRING` is returned (after being modified).

Note that the `STRING` really is modified, no copy is made!

```
(translate-string "abc" upcase-table)
=> "ABC"
```

- Variable: upcase-table  
This is a 256-character long string which may be used as a translation table to convert from lower-case to upper-case with the functions 'translate-string' and 'translate-area'.
- Variable: downcase-table  
Similar to 'upcase-table' except that it is used to convert from upper-case to lower-case.

The following functions use the translation functions and the two translation tables described above.

- Command: upcase-area START-POS END-POS &optional BUFFER  
Makes all alphabetic characters in the specified region of text upper-case.  
  
When called interactively uses the block marks for its arguments; note that this won't work properly with rectangular blocks.
- Command: downcase-area START-POS END-POS &optional BUFFER  
Similar to 'upcase-area' but makes all alphabetic characters lower-case.
- Command: upcase-word COUNT  
For the next COUNT words starting at the cursor position, make their alphabetic characters upper-case.
- Command: downcase-word COUNT  
Does the opposite of 'upcase-word', makes words lower-case!
- Command: capitalize-word  
The first character of this word (normally the one under the cursor) is made upper-case, the rest lower.

## 1.256 jade.guide/Search and Match Functions

### Searching and Matching Functions

---

The most powerful of the searching and matching functions are those using regular expressions, for details of the regexp syntax used by Jade see

#### Regular Expressions

.

Note that the regexp matcher *\*does not work across lines\**, at the moment no regexp may span more than one line. Also the regexp routines choke on NUL bytes; hopefully I'll correct these problems soon...

---

|                   |                                                        |
|-------------------|--------------------------------------------------------|
| Searching Buffers | Scanning buffers for something                         |
| String Matching   | Matching regexps to text                               |
| Replacing Strings | Replacing a found string or regexp with something else |
| Regexp Functions  | General regexp utility functions                       |

## 1.257 jade.guide/Searching Buffers

Searching Buffers  
 .....

- Function: `find-next-regexp REGEXP &optional POS BUFFER IGNORE-CASE`  
 This function returns the position of the next substring in the buffer matching the regular expression string REGEXP. It starts searching at POS, or the cursor position if POS is undefined.  
  
 If no match of the regexp occurs before the end of the buffer 'nil' is returned.  
  
 If the IGNORE-CASE argument is non-'nil' then the case of matched strings is ignored (note that character ranges are still case-significant).
  - Function: `find-prev-regexp REGEXP &optional POS BUFFER IGNORE-CASE`  
 Similar to 'find-next-regexp' except this searches in the opposite direction, from POS (or the cursor) to the \*start\* of the buffer.
  - Function: `find-next-string STRING &optional POS BUFFER`  
 Scans forwards from POS (or the cursor), in BUFFER (or the current buffer), looking for a match with the string STRING. Returns the position of the next match or 'nil'.
  - Note that matches can't span more than one line.
  - Function: `find-prev-string STRING &optional POS BUFFER`  
 A backwards-searching version of 'find-next-string'.
  - Function: `find-next-char CHARACTER &optional POS BUFFER`  
 Search forwards for an occurrence of the character CHARACTER and returns its position, or 'nil' if no occurrence exists.
  - Function: `find-prev-char CHARACTER &optional POS BUFFER`  
 This function searches backwards for an occurrence of the character CHARACTER.
-

## 1.258 jade.guide/String Matching

### String Matching

.....

- Function: `looking-at REGEXP &optional POS BUFFER IGNORE-CASE`  
Returns `'t'` if the regular expression `REGEXP` matches the text at position `POS` in the buffer `BUFFER` (or the current buffer).

Only the text from `POS` to the end of the line is matched against.

- Function: `regexp-match REGEXP STRING &optional IGNORE-CASE`  
This function returns `'t'` if the regular expression `REGEXP` matches the string `STRING`.

Note that the match is unanchored so if you want test for a match of the whole of `STRING` use the `'^'` and `'$'` regexp meta-characters. For example,

```
(regexp-match "(a|b)+" "fooabababar")
=> t
```

```
(regexp-match "^(a|b)+$" "fooabababar")
=> nil
```

```
(regexp-match "^(a|b)+$" "ababbabba")
=> t
```

When the `IGNORE-CASE` argument is non-`'nil'` the case of strings being matched is insignificant (except in character ranges).

- Function: `regexp-expand REGEXP STRING TEMPLATE &optional IGNORE-CASE`  
This function matches the regular expression `REGEXP` against the string `STRING`, if the match is successful a string is created by expanding the template string `TEMPLATE`.

For details of what meta-characters are allowed in `TEMPLATE` see

### Regular Expressions

.

```
(regexp-expand "^[a-z]+:([0-9]+)$"
"foobar:42"
"The \1 is \2.")
=> "The foobar is 42."
```

- Function: `regexp-match-line REGEXP &optional LINE-POS BUFFER IGNORE-CASE`

This function is similar to `'regexp-match'`, instead of explicitly supplying the string to match against it is one whole line of the specified buffer, the line pointed to by `LINE-POS` (or the line that the cursor is on).

't' is returned if the match is successful.

- Function: `regexp-expand-line REGEXP TEMPLATE &optional LINE-POS BUFFER IGNORE-CASE`  
As `'regexp-match-line'` is similar to `'regexp-match'`, this function is similar to `'regexp-expand'`.

The whole of the line at the position `LINE-POS` (or the cursor) is matched with the regular expression `REGEXP`. If the match is successful the `TEMPLATE` is used to expand a string which is returned.

## 1.259 jade.guide/Replacing Strings

Replacing Strings

.....

- Function: `replace-regexp REGEXP TEMPLATE &optional POS BUFFER IGNORE-CASE`  
If a substring of the buffer at `POS` (or the cursor) matches the regular expression `REGEXP` the text that matched is replaced with the result of expanding the template string `TEMPLATE`.

For details about templates see  
Regular Expressions

.

'nil' is returned if the match failed, and therefore no replacement occurred.

- Function: `replace-string OLD-STRING NEW-STRING &optional POS BUFFER`  
If a substring of the buffer at `POS` (or the cursor) matches the string `OLD-STRING` it is replaced by the string `NEW-STRING`.

If the match fails 'nil' is returned, otherwise some non-'nil' value.

## 1.260 jade.guide/Regexp Functions

Regexp Functions

.....

It is often useful to construct regular expressions by concatenating several strings together; the problem with doing this is you may not know if a string contains any characters which the regexp compiler reacts specially to (i.e. `'*'`, `'|'`, ...). Obviously these characters should be protected by a backslash, the following function will do this for you.



- Function: `regexp-quote` STRING

This function returns a new version of the string STRING, any characters in STRING which are regexp meta-characters are quoted with a backslash.

If the string contains no meta-characters the original string is returned, without being copied.

```
(regexp-quote "foo*bar+baz")
=> "foo*bar\\+baz"
```

Note that in the above example the backslashes in the returned string are only single backslashes; the print functions print a single backslash character as `'\'` so they can be read back in.

This function is usually used when a part of a regexp being constructed is unknown at compile time, often provided by the user.

As the section describing regexp syntax notes, the strings that parenthesised expressions match are recorded, the following two functions allow Lisp programs to access the positions of these strings.

- Function: `match-start` &optional EXPRESSION-INDEX

This function returns the position which the parenthesised expression number EXPRESSION-INDEX started at in the last successful regexp match.

If EXPRESSION-INDEX is `'nil'` or zero the start of the whole string matched is returned instead.

The returned value will either be a position object if the last match was in a buffer, or an integer if the last match was in a string (i.e. `'regexp-match'`).

```
(regexp-match "foo(bar)" "xyzfoobarsaalsd")
=> t
(match-start)
=> 3
(match-start 1)
=> 6
```

- Function: `match-end` &optional EXPRESSION-INDEX

Return the position which the parenthesised expression number EXPRESSION-INDEX ended at in the last successful regexp match.

If EXPRESSION-INDEX is `'nil'` or zero the end of the whole match is returned instead.

The returned value will either be a position object if the last match was in a buffer, or an integer if the last match was in a string (i.e. `'regexp-match'`).

```
(regexp-match "foo(bar)" "xyzfoobarsaalsd")
=> t
(match-end)
=> 9
```

---

```
(match-end 1)
=> 9
```

## 1.261 jade.guide/Rectangular Editing

### Rectangular Editing

---

These functions are used to manipulate rectangular regions of buffers. Two position objects are used to define a rectangle, these represent opposite corners of the rectangle. Note that the corner on the right hand side of the rectangle specifies the column *after* the last column included in the rectangle.

- Function: `delete-rect START-POS END-POS &optional BUFFER`  
This function deletes a rectangle, defined by `START-POS` and `END-POS`, from the specified buffer.
- Function: `copy-rect START-POS END-POS &optional BUFFER`  
Returns a string containing the rectangle of text defined by the two positions `START-POS` and `END-POS`. Any `TAB` characters are expanded to `SPC` characters, newline characters mark the end of each line in the rectangle.
- Function: `cut-rect START-POS END-POS &optional BUFFER`  
A combination of the `'copy-rect'` and `'delete-rect'` functions; it makes a copy of the rectangle's contents which is returned after the rectangle is deleted from the buffer.
- Command: `insert-rect STRING &optional POS BUFFER`  
Inserts the string `STRING` into the buffer at the specified position, treating `STRING` as a rectangle of text. This means that each successive line of `STRING` (separated by newline characters) is inserted at the *same* column in successive lines.

If the end of the buffer is reached and there is still some of the string left to insert extra lines are created at the end of the buffer.

- Command: `yank-rectangle &optional DONT-YANK-BLOCK`  
This function is similar to the `'yank'` function (see

Insertion Functions  
) , except that it uses the `'insert-rect'` function to insert the piece of text.

## 1.262 jade.guide/Controlling Undo

---

## Controlling Undo

---

For the description of one part of controlling the undo feature, the maximum size of the undo-list, see

Undo

.

- Variable: `buffer-record-undo`

A buffer-local variable which, when set to `'nil'`, stops any undo-information being recorded for the buffer.

When a buffer is created, this variable is always set to `'t'`.

- Variable: `buffer-undo-list`

This buffer-local variable stores the actual list of undo-information; each element defines one modification to the buffer.

Don't try to be clever and access the contents of this list; the structure may well change in future revisions of Jade.

The only thing you're allowed to do is set it to `'nil'`, this clears all undo-information for the buffer.

- Command: `undo`

Undo every change to the contents of the buffer back to the previous command. Successive calls to this command work backwards through the buffer's undo-list.

## 1.263 jade.guide/Misc Text Functions

### Miscellaneous Text Functions

---

- Function: `empty-line-p` &optional POS BUFFER

This function returns `'t'` if the line pointed to by POS (or by the cursor) consists totally of TAB or SPC characters.

## 1.264 jade.guide/Writing Modes

### Writing Modes

---

Modes are used to customise individual buffers so that the text it contains can be edited in a special way. Each buffer has a single "Major mode", tailoring the buffer to the type of file contained in it (i.e. C source code uses `'c-mode'`). See

---

## Editing Modes

.

"Minor modes" provide individual features which may be enabled and disabled individually, each buffer may have any number of minor modes enabled at once. See

## Minor Modes

.

## Writing Major Modes

How to define a new major mode

## Installing Modes

Functions and variables used to  
install major modes in buffers

## Writing Minor Modes

Minor modes are totally different  
to major modes

## Mode-Specific Indentation

Each major mode may define its own  
method of indentation,

## Mode-Specific Expressions

expression handling,

## Mode-Specific Comments

and comment insertion.

## 1.265 jade.guide/Writing Major Modes

### Writing Major Modes

-----

Each major mode must define a command whose name ends in `'-mode'` (i.e. `'c-mode'`, `'lisp-mode'`, etc...). This command is called when the major mode is to be installed in the current buffer. It's first action *must* be to check for an already installed mode and remove it. The following code fragment does this,

```
(when major-mode-kill
 (funcall major-mode-kill))
```

*\*All\* major modes must do this!*

Now the major mode is free to install itself; generally this will entail setting the buffer-local values of the `'mode-name'`, `'major-mode'`, `'major-mode-kill'` and `'keymap-path'` variables. For example the `'lisp-mode'` sets these variables as follows,

```
(setq mode-name "Lisp"
 major-mode 'lisp-mode
 major-mode-kill 'lisp-mode-kill
 keymap-path (cons 'lisp-mode-keymap keymap-path))
```

Note how the major mode's own keymap (with all the mode's local key bindings installed in it) is consed onto the front of the 'keymap-path'; this ensures that mode-local bindings take precedence over bindings in the global keymaps.

After installing itself a major mode should call a hook (generally called 'X-mode-hook' where X is the name of the mode) to allow customisation of the mode itself.

The 'major-mode-kill' variable holds a function to be called when the major mode is to be removed from the current buffer; basically it should remove its keymap and set all the mode-local variables to 'nil'. For example the 'lisp-mode-kill' function does the following to negate the effects of the code fragment above,

```
(setq keymap-path (delq 'lisp-mode-keymap keymap-path)
 major-mode nil
 major-mode-kill nil
 mode-name nil)
```

- Variable: major-mode

This buffer-local variable contains the symbol whose function definition was used to install the buffer's major mode (i.e. 'c-mode', etc...).

When it is 'nil' the buffer uses the 'generic' mode; this is simply the bog standard editor.

- Variable: major-mode-kill

This buffer-local variable contains the function which should be called to remove the buffer's currently installed major-mode.

Note that the 'kill-buffer' function calls this (if it's non-'nil') just before destroying a buffer; so if necessary, an error signalled within this function will prevent a buffer being killed.

- Variable: mode-name

A buffer-local variable containing the 'pretty' name of the buffer's major mode, a string which will be printed in the status line.

Many modes bind commands to keys with the prefix 'Ctrl-c', to save each mode creating a new root keymap the buffer-local variable 'ctrl-c-keymap' exists.

- Variable: ctrl-c-keymap

This buffer-local variable can be used by major modes to hang their keymap for the 'Ctrl-c' prefix from. Simply set this variable to the keymap your mode wants to be installed after a 'Ctrl-c' prefix.

The definitions for many different types of modes can be found in Jade's lisp directory.

---

## 1.266 jade.guide/Installing Modes

### Installing Modes

---

Before a major mode can be used to edit a buffer with it must be installed in that buffer. The most straightforward method of doing this is simply to invoke the mode's command which does this (i.e. 'c-mode').

It could be a bit annoying to have to do this every time a new buffer is created so the 'mode-alist' variable allows major modes to be installed automatically, when the buffer is opened.

- Function: `init-mode BUFFER &optional STRING`

This function attempts to install a major mode into BUFFER. If the 'major-mode' variable is non-'nil' it defines the function to call to install the mode; this function will be called.

Otherwise the 'mode-alist' variable is searched; each regular expression is matched against a string, when a match occurs the associated function is called to install the mode.

The string matched against is defined by the first of the following choices which is not 'nil' or undefined.

1. The value of the optional STRING argument.
2. The word specified on the first line of the buffer bracketed by the string '-\*-'. For example if the first line contained the string '-\*-Text-\*-' the string 'Text' would be used.
3. The value of the variable mode-name.
4. The name of the file being edited in the buffer.

Note that each match is case-insensitive.

- Variable: `mode-alist`

An association list (see Association Lists) defining regular expressions which associate with a particular major mode.

When the 'init-mode' function matches a regular expression to the string it is using to find the mode for the buffer the associated mode is installed.

For example, 'mode-alist' could be,

```
(("\.(c|h)$|^c(|-mode)$" . c-mode)
 ("\.jl$|^lisp(|-mode)$" . lisp-mode)
 ("\. (text|doc|txt|article|letter)$" . text-mode)
 ("^(text(|-mode)|(.*/|)draft)$" . text-mode)
```

---

```
("^indented-text(|-mode)$" . indented-text-mode)
("\.[s]$|^asm(|-mode)$" . asm-mode)
("\.[S]$|^asm-cpp(|-mode)$" . asm-cpp-mode)
("\.texi(|nfo|^texinfo(|-mode)$" . texinfo-mode))
```

- Function: kill-mode &optional BUFFER  
This function removes the major mode currently installed in the specified buffer.

## 1.267 jade.guide/Writing Minor Modes

### Writing Minor Modes

-----

Minor modes are generally harder to write properly than major modes since they have to peacefully coexist with all the other minor modes which may also be enabled in a buffer.

Generally each minor mode maintains a buffer-local variable saying whether or not it's installed in the buffer. The minor mode's function usually toggles the mode on or off depending on the state of this variable.

There are two functions which *must* be used to install and remove a minor mode -- 'add-minor-mode' and 'remove-minor-mode', see their documentation for details.

Each buffer has a keymap containing the bindings of all the minor modes enabled in the buffer (the variable 'minor-mode-keymap'). These bindings have to be added when the mode is enabled and removed when it is disabled.

- Variable: minor-mode-list  
This buffer-local variable is a list of all the minor modes enabled in a buffer.
- Variable: minor-mode-names  
This buffer-local variable contains a list of strings, each string names one of the minor modes currently enabled in the buffer.
- Variable: minor-mode-keymap  
A buffer-local keymap to be used by minor-modes. This is only created the first time a minor mode calls 'add-minor-mode' in the buffer.
- Function: add-minor-mode MODE NAME &optional NO-KEYMAP  
This function installs a minor mode (the symbol MODE) into the current buffer. All minor modes should call this before doing anything drastic.

NAME is the string to be displayed in the status line as the name of this minor mode.

---

When NO-KEYMAP is 'nil' or undefined this function ensures that the 'minor-mode-keymap' variable has a valid value in this buffer.

- Function: remove-minor-mode MODE NAME  
Removes a minor mode from the current buffer, the MODE and NAME arguments must have the same value as the arguments given to 'add-minor-mode' when the mode was enabled.

The following code fragment is an example minor mode taken from Jade's source code.

```
(provide 'fill-mode)

(defvar fill-column 72
 "Position at which the text filling commands break lines.")

(defvar fill-mode-p nil)
(make-variable-buffer-local 'fill-mode-p)

;;;###autoload
(defun fill-mode ()
 "Minor mode for automatically filling lines, i.e. word-wrapping.
This makes the SPC key checks if the cursor is past the fill-column. If
so, the next line is started."
 (interactive)
 (if fill-mode-p
 (progn
 (setq fill-mode-p nil)
 (remove-minor-mode 'fill-mode "Fill")
 (unbind-keys minor-mode-keymap "SPC"))
 (add-minor-mode 'fill-mode "Fill")
 (setq fill-mode-p t)
 (bind-keys minor-mode-keymap
 "SPC" 'fill-mode-spc)))

(defun fill-mode-spc ()
 (interactive)
 (when (> (pos-col (cursor-pos)) fill-column)
 (let
 ((pos (cursor-pos)))
 (set-pos-col pos (1+ fill-column))
 (setq pos (unless (word-start pos) (forward-word -1 pos)))
 (insert "\n" pos)
 (let
 ((end (left-char 1 (copy-pos pos))))
 (when (equal (get-char end) ?\)
 (delete-area end pos))))))
 (insert " "))
```

## 1.268 jade.guide/Mode-Specific Indentation

Mode-Specific Indentation

---



Some major modes provide functions which manage the indentation of the buffer they are installed in. These modes are usually those which are designed for a particular programming language; for example C mode understands how to indent C source and Lisp mode knows about Lisp code.

To simplify matters there is a unified interface to the indentation process; each major mode simply sets the value of a buffer-local variable to the function used to indent a line in that buffer. This variable is then referenced by the functions which provide indentation.

- Variable: mode-indent-line

This buffer-local variable should contain a function when the buffer's major mode provides special indentation.

The function should take one optional argument, the position of the line to indent. If the value of this argument is 'nil' the current line should be indented. The function should set the indentation of the line to the correct depth then return the glyph position (see

Glyph Positions  
) of the first non-whitespace

character.

For example Lisp mode sets this variable to 'lisp-indent-line', this function is defined as,

```
(defun lisp-indent-line (&optional pos)
 (set-indent-pos (lisp-indent-pos (or pos (cursor-pos))))))
```

Where the function 'lisp-indent-pos' calculates the proper indentation for the line pointed to by its argument.

For the functions dealing with indentation see  
Indentation Functions

.

## 1.269 jade.guide/Mode-Specific Expressions

Mode-Specific Expressions  
-----

Most programming use the concept of an "expression", Jade allows major modes to define two functions which define the syntax of an expression in a particular programming language. Commands exist which use these functions to allow the manipulation of expressions as entities in a buffer, much like words.

- Variable: mode-forward-exp

This buffer-local variable contains a function which calculates the position of the end of an expression in that language.

The lambda-list of the function (i.e. its arguments) must be '(&optional COUNT POS)'. COUNT is the number of expressions to

move forwards over (default is one), POS is the position to start from (default is the cursor position).

The function should return the position of the character following the end of COUNT expressions starting from POS.

- Variable: mode-backward-exp  
Similar to 'mode-forward-exp' but works backwards from the character after the expression (at POS) to the start of the previous COUNT expressions.

These functions can often be quite complex but their structure is usually the same; these two examples are taken from the Lisp mode,

```
(defun lisp-forward-sexp (&optional number pos)
 "Return the position of the NUMBER'th next s-expression from POS."
 (unless number
 (setq number 1))
 (while (> number 0)
 ;; Move 'pos' over one expression
 ...
 (setq number (1- number)))
 pos)
```

```
(defun lisp-backward-sexp (&optional number orig-pos)
 "Return the position of the NUMBER'th previous s-expression
from ORIG-POS."
 (unless number
 (setq number 1))
 (unless orig-pos
 (setq orig-pos (cursor-pos)))
 (let
 ((pos (copy-pos orig-pos)))
 (while (> number 0)
 ;; Move 'pos' backwards over one expression
 ...
 (setq number (1- number)))
 pos))
```

## 1.270 jade.guide/Mode-Specific Comments

### Mode-Specific Comments

---

When you wish to enter a comment in a piece of source code Jade has a command to do this ('insert-comment'); each major mode which wishes to allow comments (created by this command) must give the following variable a suitable function.

- Variable: mode-comment-fun  
This buffer-local variable contains the function to call when a comment is to be entered, basically the 'insert-comment' command just calls this function.
-

- Function: `find-comment-pos`  
This function moves the cursor to a suitable position for inserting a comment in the current line.
- Variable: `comment-column`  
Buffer-local variable containing the canonical column number which comments should begin at (used by the `'find-comment-pos'` function). If the line extends past this column the next tab stop after the end of the line is used instead.

The following function is an example of what is needed in the `'mode-comment-fun'` variable; it is used by the C mode.

```
(defun c-insert-comment ()
 (interactive)
 (find-comment-pos)
 (insert "/* */")
 (goto-left-char 3))
```

## 1.271 jade.guide/Prompting

### Prompting

=====

The most common way to ask the user for a response is to encode the question in the command's interactive declaration (see

#### Interactive Declarations

), sometimes this is inconvenient; functions are available which have the same effect as the code letters in an interactive declaration.

The following two functions don't have an equivalent code for the interactive declaration.

- Function: `y-or-n-p QUESTION`  
This function prompts the user for a single key response to the string `QUESTION` asking a question which can be answered yes or no.  
  
Returns `'t'` when `QUESTION` is answered with a `'y'` and `'nil'` when `'n'` is typed.
- Function: `yes-or-no-p QUESTION`  
Similar to `'y-or-n-p'` but the answer must be either the word `'yes'` or the word `'no'` entered in full. This function should be used when a mistyped answer could be catastrophic (i.e. losing changes to a buffer).  
  
Returns `'t'` for `'yes'`, `'nil'` for anything else.

The following functions are the functions used by the `'call-command'` function to resolve interactive arguments.

Note that these function don't return the string entered (except for

'prompt-for-string') -- they return some Lisp object which the string entered represents somehow.

- Function: `prompt-for-file` &optional PROMPT EXISTING START  
Prompts for the name of a file. PROMPT is the string to display at the head of the prompt, when EXISTING is non-'nil' only files which actually exist are allowed to be entered. The START argument may be a string defining the starting contents of the prompt.
- Function: `prompt-for-directory` &optional PROMPT EXISTING START  
Prompts for the name of a directory, all arguments are similar to in the 'prompt-for-file' function.
- Function: `prompt-for-buffer` &optional PROMPT EXISTING DEFAULT  
This function prompts for a buffer object, if EXISTING is non-'nil' the buffer selected must exist, otherwise the buffer will be created if it doesn't already exist. DEFAULT is the value to return if the user enters the null string, if 'nil' the current buffer is returned.

Note that this returns the *\*actual buffer\**, not its name as a string.

- Function: `prompt-for-symbol` &optional PROMPT PREDICATE  
Prompt for a symbol, PROMPT is displayed at the head of the prompt buffer. If the PREDICATE argument is defined it is a predicate function; only symbols which when applied to the function PREDICATE return non-'nil' will be allowed to be entered.
- Function: `prompt-for-lisp` &optional PROMPT  
Prompt for and return a Lisp object.
- Function: `prompt-for-function` &optional PROMPT  
Prompts for a function.

```
(prompt-for-function PROMPT)
==
(prompt-for-symbol PROMPT 'functionp)
```

- Function: `prompt-for-variable` &optional PROMPT  
Prompts for a variable (a symbol whose value is not void).

```
(prompt-for-variable PROMPT)
==
(prompt-for-symbol PROMPT 'boundp)
```

- Function: `prompt-for-command` &optional PROMPT  
Prompts for a command (a function which may be called interactively).

```
(prompt-for-command PROMPT)
==
(prompt-for-symbol PROMPT 'commandp)
```

- Function: `prompt-for-string` &optional PROMPT  
Prompt for a string, whatever string is entered is returned as-is.

- Function: `prompt-for-number` &optional PROMPT  
Prompts for a number which is then returned.

The following function is useful when a number of options have to be chosen between, for example the menu command in Info-mode uses this function.

- Function: `prompt-from-list` OPTION-LIST PROMPT &optional START  
Returns a selected choice from the list of options (strings) OPTION-LIST. PROMPT is the title displayed, START the optional starting choice.

## 1.272 jade.guide/Files

### Files

=====

Jade allows you to manipulate files in the operating system's filing system; a special type of Lisp object, a "file object", is used to represent files which have been opened for reading or writing (through the streams mechanism, see

Streams  
)

Names of files are represented by strings, the syntax of file names is defined by the underlying operating system: Jade simply treats it as a string.

#### File Names

Files are named by a string

#### File Objects

Lisp objects representing files

#### File Information

Predicates on files

#### Manipulating Files

Deleting, renaming and copying files

#### Reading Directories

Getting a list of the files in a directory

#### Reading and Writing Files

Accessing the contents of a file in one go

---

## 1.273 jade.guide/File Names

### File Names

-----

A "file name" is a string identifying an individual file (or directory) in the filing system (i.e. the disk). The exact syntax of file names depends on the operating system.

- Function: file-name-directory FILE-NAME

This function returns the directory part of the file name string FILE-NAME. This is the substring of FILE-NAME defining the directory containing the file.

```
(file-name-directory "/tmp/foo")
```

```
=> "/tmp/"
```

```
(file-name-directory "foo")
```

```
=> ""
```

```
(file-name-directory "foo/bar/")
```

```
=> "/foo/bar/"
```

- Function: file-name-nondirectory FILE-NAME

Returns the substring of the file name FILE-NAME which is *\*not\** the directory part.

```
(file-name-nondirectory "/tmp/foo")
```

```
=> "foo"
```

```
(file-name-nondirectory "foo")
```

```
=> "foo"
```

```
(file-name-nondirectory "foo/bar/")
```

```
=> ""
```

- Function: file-name-concat &rest PARTS

This function returns a file name constructed by concatenating each of the PARTS of the file name together. Each part is separated by the necessary string (i.e. '/' on Unix) when necessary. Note that each part may contain more than one component of the file name.

```
(file-name-concat "/tmp" "foo" "bar")
```

```
=> "/tmp/foo/bar"
```

```
(file-name-concat "/tmp/" "foo/" "bar")
```

```
=> "/tmp/foo/bar"
```

```
(file-name-concat "/tmp/foo" "bar")
```

```
=> "/tmp/foo/bar"
```

- Function: expand-file-name FILE-NAME &optional MAKE-ABSOLUTE

This function expands the string FILE-NAME into a valid file name. Currently it only checks for a leading tilde character ('~') when running on Unix, if one is found it's expanded to the user's home

directory.

When the optional argument MAKE-ABSOLUTE is non-`'nil'` FILE-NAME is altered so that it is not relative to the current working directory. Generally this involves prefixing it by the absolute name of the current directory.

```
(expand-file-name "~/src")
=> "/home/jsh/src"
```

```
(expand-file-name "foo.c" t)
=> "/var/src/jade/foo.c"
```

- Function: `tmp-file-name`

This function returns the name of a file which, when created, may be used for temporary storage. Each time this function is called a unique name is computed.

```
(tmp-file-name)
=> "/tmp/00088aaa"
```

```
(tmp-file-name)
=> "/tmp/00088baa"
```

## 1.274 jade.guide/File Objects

### File Objects

-----

A file object is a Lisp object which represents a file in the filing system. Any file object may be used as a stream (either input or output) to access the contents of the file serially,

Streams

.

Creating File Objects

Opening files

Destroying File Objects

Closing files

File Object Predicates

Predicates for file objects

Functions on File Objects

Functions operating on file objects

---

## 1.275 jade.guide/Creating File Objects

### Creating File Objects

.....

- Function: open FILE-NAME MODE-STRING &optional FILE-OBJECT

This function opens the file called FILE-NAME (see  
File Names  
) and  
returns the file's object.

The MODE-STRING argument is a string defining the access modes used to open the file with; this string is passed as-is to the C library's 'fopen()' function. Usually one of the following strings is used,

'r'

Open an existing file for reading only.

'w'

Open the file for writing only, if the file exists it is truncated to zero length. Otherwise a new file is created.

'a'

Open the file for appending to, i.e. writing to the end of the file. If the file doesn't exist it is created.

Other options exist; consult a C library manual for details.

When the FILE-OBJECT argument is defined it should be a file object, the file it points to will be closed and the new file will be opened on this object.

## 1.276 jade.guide/Destroying File Objects

### Destroying File Objects

.....

The easiest way to close a file is simply to eliminate all references to it, subsequently the garbage collector will close it for you. It is better to close files explicitly though since only a limited number of files may be opened concurrently.

- Function: close FILE-OBJECT

This function closes the file pointed to by the file object  
FILE-OBJECT.

Until a new file is opened on FILE-OBJECT any read/write accesses to it are illegal and an error will be signalled.



## 1.277 jade.guide/File Object Predicates

File Object Predicates

.....

- Function: filep OBJECT  
This function returns 't' when its argument is a file object.
- Function: file-bound-p FILE-OBJECT  
Returns 't' when the file object FILE-OBJECT is currently bound to a physical file (i.e. the 'close' function hasn't been called on it yet).
- Function: file-eof-p FILE-OBJECT  
This function returns 't' when the current position of the file object FILE-OBJECT is the end of the file (i.e. when reading a character from the file would return 'nil').

## 1.278 jade.guide/Functions on File Objects

Functions on File Objects

.....

- Function: flush-file FILE-OBJECT  
This function flushes any buffered output to the file object FILE-OBJECT to disk.  
  
Note that when using a file which was opened with the '+' option it's necessary to call this function when switching from reading to writing or vice versa.
- Function: file-binding FILE-OBJECT  
Returns the name of the file which the file object FILE-OBJECT is currently bound to.
- Function: read-file-until FILE-OBJECT REGEXP &optional IGNORE-CASE  
This function reads lines from the file object FILE-OBJECT until a line matching the regular expression REGEXP is found. The matching line is returned, or 'nil' if the end of the file is reached.  
  
When the IGNORE-CASE option is non-'nil' all regexp matching is done case-insignificantly (except for matching ranges).

## 1.279 jade.guide/File Information

File Information

-----

A number of functions exist which when given the name of a file return some information about that file.

- Function: `file-exists-p FILE-NAME`  
Returns `'t'` when a file `FILE-NAME` exists.
- Function: `file-regular-p FILE-NAME`  
Returns `'t'` when the file `FILE-NAME` is a 'normal' file. This means that it isn't a directory, device, symbolic link or whatever.
- Function: `file-directory-p FILE-NAME`  
Returns `'t'` when the file `FILE-NAME` is a directory.
- Function: `file-symlink-p FILE-NAME`  
Returns `'t'` when the file `FILE-NAME` is a symbolic link.
- Function: `file-readable-p FILE-NAME`  
Returns `'t'` when the file `FILE-NAME` is readable.
- Function: `file-writable-p FILE-NAME`  
Returns `'t'` when the file `FILE-NAME` is writable.
- Function: `file-owner-p FILE-NAME`  
Returns `'t'` when the ownership of the file `FILE-NAME` is the same as that of any files written by the editor.

Note that currently this always returns `'t'` in the Amiga version.

- Function: `file-nlinks FILE-NAME`  
Returns the number of hard links pointing to the file `FILE-NAME`. If `FILE-NAME` has only one name the number will be one.

Note that this always returns one in the Amiga version of Jade.

- Function: `file-modes FILE-NAME`  
This function returns the access permissions of the file `FILE-NAME`. This will be an integer whose format is undefined; it differs from operating system to operating system.
- Function: `set-file-modes FILE-NAME MODES`  
This function sets the access permissions of the file `FILE-NAME` to the integer `MODES` (as returned by the `'file-modes'` function).
- Function: `file-modtime FILE-NAME`  
Returns the system time at the last modification to the file `FILE-NAME`, this will be an integer. See  
System Time
- Function: `file-newer-than-file-p FILE-NAME1 FILE-NAME2`  
This function returns `'t'` if the file `FILE-NAME1` was modified more recently than the file `FILE-NAME2` was.

```
(file-newer-than-file-p FILE1 FILE2)
==
(> (file-modtime FILE1) (file-modtime FILE2))
```

## 1.280 jade.guide/Manipulating Files

### Manipulating Files

---

- Command: delete-file FILE-NAME  
This function deletes the file called FILE-NAME. When called interactively FILE-NAME is prompted for.
  
- Command: rename-file FILE-NAME NEW-NAME  
This function attempts to change the name of the file NEW-NAME to NEW-NAME.  
  
This won't work from one file system to another or if a file called NEW-NAME already exists, in these cases an error is signalled.  
  
This prompts for its arguments when called interactively.
  
- Command: copy-file FILE-NAME DESTINATION-NAME  
Creates a new copy of the file FILE-NAME with the name DESTINATION-NAME.  
  
The access modes of the new file will be the same as those of the original file.  
  
The arguments are prompted for when this function is called interactively.

## 1.281 jade.guide/Reading Directories

### Reading Directories

---

- Function: directory-files DIRECTORY-NAME  
This function returns a list of the names of all files in the directory whose file name is DIRECTORY-NAME. The names in the list will be relative to the directory DIRECTORY-NAME, any directories in the list will have a '/' character appended to them.

```
(directory-files "/tmp/foo"
=> ("bar" "subdir/" "xyz" "." ".."))
```

## 1.282 jade.guide/Reading and Writing Files

### Reading and Writing Files

---

- Function: read-file FILE-NAME  
This function returns a string containing the contents of the file called FILE-NAME.
-

- Function: write-file FILE-NAME CONTENTS  
This function creates or overwrites the file called FILE-NAME with the string CONTENTS as its contents.

## 1.283 jade.guide/Processes

### Processes

=====

When running on a Unix-style operating system (i.e. the X11 version) Jade allows you to launch and control an arbitrary number of subprocesses. These subprocesses can run either synchronously or asynchronously in respect to the editor; data can be sent to the stdin channel and any output from the process is automatically written to a programmer-defined Lisp stream.

Currently there is *\*no\** way to manipulate subprocesses in the Amiga version of Jade (sorry!).

#### Process Objects

Lisp objects associated with subprocesses

#### Asynchronous Processes

Subprocesses running in parallel with Jade

#### Synchronous Processes

Subprocesses which Jade runs serially

#### Process I-O

Input and output with subprocesses

#### Process States

Suspending subprocesses

#### Signalling Processes

Sending signals to subprocesses

#### Process Information

Information stored in a process object

#### Interactive Processes

Shell mode lets the user interact with a subprocess

## 1.284 jade.guide/Process Objects

---

## Process Objects

---

A "process object" is a type of Lisp object used to provide a link between a 'physical' process running in the operating system and Jade's Lisp system. Each process object consists of a number of components (references to other Lisp objects); these components are used when the object is used to run a subprocess.

Process objects which aren't currently being used to run a subprocess store the exit value of the last subprocess which was run on that object.

- Function: `processp` OBJECT

This function returns 't' when its argument is a process object.

The programmer-accessible components of a process object are,

"Output stream"

A normal Lisp output stream (see `Output Streams`), all data which the subprocess outputs to its 'stdout' channel is copied to this output stream. See `Process I-O`.

"State change function"

A Lisp function, called each time the state of the subprocess being run on the object changes. See `Process States`.

"Program name"

The name of the program (a string) to execute when the subprocess is created.

"Program arguments"

A list of strings defining the arguments which the program executed is given.

"Directory"

When a subprocess is started its current working directory is set to the directory named by this component of its process object.

"Connection type"

Asynchronous subprocesses (see `Asynchronous Processes`) use this component to decide how to connect to the I/O channels of the subprocess. Current options include pseudo-terminals and pipes.

- Function: `make-process` &optional OUTPUT-STREAM STATE-FUNCTION DIRECTORY PROGRAM ARGS

This functions creates and returns a new process object. \*No subprocess will be started.\*

---

The optional arguments are used to define the values of the components of the new process object, any undefined components will be set to default or null values.

For each component of a process object two functions exist; one to read the component's value in a specific process object, the other to set the component's value.

- Function: `process-prog PROCESS`  
Returns the value of the program name component of the process object `PROCESS`.
- Function: `set-process-prog PROCESS PROG-NAME`  
Sets the value of the program name component of the process object `PROCESS` to the string `PROG-NAME`, then returns `PROG-NAME`.
- Function: `process-args PROCESS`  
Returns the value of the program arguments component of the process object `PROCESS`.
- Function: `set-process-args PROCESS ARG-LIST`  
Sets the value of the program arguments component of the process object `PROCESS` to the list `ARG-LIST`, then returns `ARG-LIST`.
- Function: `process-dir PROCESS`  
Returns the value of the directory component of the process object `PROCESS`.
- Function: `set-process-directory PROCESS DIRECTORY`  
Sets the value of the directory component of the process object `PROCESS` to the string `DIRECTORY`, then returns `DIRECTORY`.

## 1.285 jade.guide/Asynchronous Processes

### Asynchronous Processes

---

An "asynchronous process" is one that runs in parallel with the editor, basically this means that once the subprocess has been started (by the `'start-process'` function) Jade will carry on as normal.

The event loop checks for output from asynchronous processes, any found is copied to the process' output stream, and calls the the process' state change function when necessary (see `Process States`).

When using asynchronous processes you have a choice as to the Unix mechanism used to connect the `'stdin'`, `'stdout'` and `'stderr'` streams of the subprocess to Jade's process (note that whatever the choice `'stdout'` and `'stderr'` always go to the same place).

---

The two options currently available are pipes or pseudo-terminals; in general pseudo-terminals should only be used to provide a direct interface between the user and a process (i.e. the `*shell*` buffer) since they allow job control to work properly. At other times pipes will be more efficient and are used by default.

- Function: `start-process` &optional PROCESS-OBJECT PROGRAM &rest ARGS  
This function starts an asynchronous subprocess running on the process object PROCESS-OBJECT. If PROCESS-OBJECT is undefined a new process object is created (by calling the function `'make-process'` with all arguments undefined).

The function always returns the process object which the subprocess has been started on. If for some reason the subprocess can't be created an error of type `'process-error'` is signalled.

The optional argument PROGRAM is a string defining the name of the program to execute, it will be searched for in all the directories in the `'PATH'` environment variable. The ARGS are strings to pass to the subprocess as its arguments.

When defined, the optional arguments overrule the values of the related components of the process object.

The following example runs the `'ls'` program asynchronously, its output is inserted into the current buffer.

```
(let
 ((process (make-process (current-buffer))))
 (start-process process "ls" "-s"))
```

Note that when Jade terminates it kills all of its asynchronous subprocesses which are still running without warning.

- Function: `process-connection-type` PROCESS  
Returns the value of the connection type component of the process object PROCESS. See the documentation of the `'set-process-connection-type'` function for the values this may take.
- Function: `set-process-connection-type` PROCESS SYMBOL  
Sets the value of the connection type component of the process object PROCESS to SYMBOL, then returns SYMBOL.

SYMBOL should be one of the following symbols,

`'pty'`

Use pseudo-terminals to connect to subprocesses running asynchronously on this process object.

`'pipe'`

Use standard Unix pipes to connect, this is the default value of this component.

---

## 1.286 jade.guide/Synchronous Processes

### Synchronous Processes

---

When a "synchronous process" is started Jade waits for it to terminated before continuing; they are usually used when a Lisp program must invoke an external program as part of its function, i.e. the auto-compression feature runs the compression program 'gzip' synchronously when it needs to compress a buffer.

Unlike asynchronous processes there is no choice between pipes and pseudo-terminals for connecting to a subprocess. Instead, it is possible to link the 'stdin' channel of a synchronous process to a named file.

- Function: run-process &optional PROCESS-OBJECT INPUT-FILE-NAME  
PROGRAM &rest ARGS

This function starts a process running on the process object PROCESS-OBJECT. If PROCESS-OBJECT is undefined a new process object is created by calling the 'make-process' function.

If defined, the string INPUT-FILE-NAME names the file to connect to the standard input of the subprocess, otherwise the subprocess' input comes from the null device ('/dev/null').

The optional arguments PROGRAM and ARGS define the name of the program to invoke and any arguments to pass to it. The program will be searched for in all directories listed in the 'PATH' environment variable.

If any of the optional parameters are unspecified they should have been set in the PROCESS-OBJECT prior to calling this function.

After successfully creating the new subprocess, this function simply copies any output from the process to the output stream defined by the output stream component of the process object. When the subprocess exits its exit-value is returned (an integer). Note that the exit-value is the value returned by the 'process-exit-value' function, see  
Process Information

If, for some reason, the new subprocess can't be created an error of type 'process-error' is signalled.

The following function definition is taken from the 'gzip.jl' file, it shows how the 'run-process' function can be used to uncompress a file into a buffer.

```
;; Uncompress FILE-NAME into the current buffer
(defun gzip-uncompress (file-name)
 (let
 ((proc (make-process (current-buffer))))
 (message (concat "Uncompressing \"" file-name "\"") t)
 ;; gunzip can do .Z files as well
 (unless (zerop (run-process proc nil "gunzip" "-c" file-name))
```



```
(signal 'file-error (list "Can't gunzip file" file-name))))))
```

## 1.287 jade.guide/Process I-O

### Process I/O

-----

It is only possible for lisp programs to explicitly send input data to *asynchronous* processes (by the time it's possible to call a function to send data to a synchronous process, the process will already have terminated!). Simply use the process object which an asynchronous process is running on as a normal Lisp input stream, any strings or characters written to the stream will immediately be copied to the 'stdin' channel of the subprocess.

With synchronous processes, the only control over input data possible is by giving the 'run-process' function the name of a file containing the subprocess' input data.

Output data from subprocesses is handled the same way by both asynchronous and synchronous processes: it is simply copied to the stream defined by the output stream component of the subprocess' process object.

- Function: process-output-stream PROCESS  
Returns the value of the output stream component of the process object PROCESS.
- Function: set-process-output-stream PROCESS STREAM  
Sets the value of the output stream component of the process object PROCESS to the stream STREAM, then returns STREAM.

See

Streams

.

## 1.288 jade.guide/Process States

### Process States

-----

Each process object has a "state" associated with it; this depends on the status of the subprocess currently running on the process object (or not as the case may be).

The possible states are,

"running"

---

This state means that the subprocess using this process object is currently running, i.e. it hasn't been stopped.

"stopped"

Means that the subprocess has been temporarily suspended from running.

"unused"

This means that the process object is free to have a new subprocess created on it.

Predicates exist which test whether a given process object is in one of these states.

- Function: `process-running-p PROCESS-OBJECT`  
Returns 't' when PROCESS-OBJECT is in the running state.
- Function: `process-stopped-p PROCESS-OBJECT`  
Returns 't' when PROCESS-OBJECT is in the stopped state.
- Function: `process-in-use-p PROCESS-OBJECT`  
Returns 't' when PROCESS-OBJECT is *\*not\** in the unused state.

The following two functions are used to stop and then subsequently continue a process running.

- Function: `stop-process PROCESS-OBJECT &optional WHOLE-GROUP`  
This function suspends execution of the subprocess running on the process object PROCESS-OBJECT.  
  
If WHOLE-GROUP is non-'nil' all subprocesses in the process group of PROCESS-OBJECT are stopped.
- Function: `continue-process PROCESS-OBJECT &optional WHOLE-GROUP`  
Use this function to continue a subprocess executing after it has been stopped (by the 'stop-process' function).  
  
If WHOLE-GROUP is non-'nil' all subprocesses in the process group of PROCESS-OBJECT are continued.

The state change function component of a process object defines a function which will be called each time the state of the process object changes. If your program needs to be informed when an asynchronous process terminates this function is the way to do it.

- Function: `process-function PROCESS`  
Returns the value of the state change function component of the process object PROCESS.
  - Function: `set-process-function PROCESS FUNCTION`  
Sets the value of the state change function component of the process object PROCESS to the function FUNCTION, then returns FUNCTION.
-

## 1.289 jade.guide/Signalling Processes

### Signalling Processes

-----

- Function: `signal-process PROCESS-OBJECT SIGNAL-NUMBER &optional WHOLE-GROUP`

If the process object `PROCESS-OBJECT` is being used to run an asynchronous subprocess send the signal numbered `SIGNAL-NUMBER` to it.

When the optional argument `WHOLE-GROUP` is non-`'nil'` the signal is also sent to all processes in the process group of the subprocess.

The following functions use the `'signal-process'` function to send some common signals to processes.

- Function: `interrupt-process PROCESS-OBJECT &optional WHOLE-GROUP`  
Sends the `'SIGINT'` signal to `PROCESS-OBJECT`.

```
(interrupt-process PROCESS-OBJECT WHOLE-GROUP)
==
(signal-process PROCESS-OBJECT 'SIGINT' WHOLE-GROUP)
```

- Function: `kill-process PROCESS-OBJECT &optional WHOLE-GROUP`  
Sends the `'SIGKILL'` signal to the `PROCESS-OBJECT`.

```
(kill-process PROCESS-OBJECT WHOLE-GROUP)
==
(signal-process PROCESS-OBJECT 'SIGKILL' WHOLE-GROUP)
```

Note that the functions `'stop-process'` and `'continue-process'` also send signals to the subprocess.

## 1.290 jade.guide/Process Information

### Process Information

-----

- Function: `process-id PROCESS-OBJECT`  
This function returns the operating-system identifier associated with the subprocess currently running on the process object `PROCESS-OBJECT`.

- Function: `process-exit-value PROCESS-OBJECT`  
Returns the integer representing the return code of the last subprocess to be run on `PROCESS-OBJECT`.

If no subprocess has been run on `PROCESS-OBJECT`, `PROCESS-OBJECT` is currently in the running state or the last subprocess exited abnormally (i.e. from a terminal signal) `'nil'` is returned.

- Function: `process-exit-status PROCESS-OBJECT`
-

This function returns the integer that was the exit status of the last subprocess which was run on the process object PROCESS-OBJECT.

Note that the exit status is *\*not\** the value given to the 'exit' function in a C program, use the 'process-exit-value' to access this value.

If no process has been run on PROCESS-OBJECT, or the process is currently in the running state 'nil' is returned.

## 1.291 jade.guide/Interactive Processes

### Interactive Processes

---

The Shell mode is usually used to run a shell process in a buffer (with the 'shell' command, see

Shell

) but in actual fact it is capable

of running (nearly) any type of interactive process. For example the gdb interface (see

Debugging Programs

) uses the Shell mode to handle

its user interaction.

The following buffer-local variables control the Shell mode.

- Variable: shell-program  
This variable defines the name of the program to execute. By default it is the user's shell.
- Variable: shell-program-args  
A list of arguments which should be given to the process when it is started.
- Variable: shell-prompt-regexp  
This regular expression must match the prompt that the process emits each time it waits for input. Its standard value of '^([^\#%>])\*([^\#%>]) \*' will need to be tailored to the program that you are executing.
- Variable: shell-callback-function  
Every time the state of the subprocess changes (see Process States)  
)  
this function is called in the context of the process' buffer.
- Variable: shell-output-stream  
All output from the subprocess is copied to this output stream. If it is 'nil' all output goes to the end of the process' buffer.

Note that this variable is only referenced when the process is started.

---

To use the Shell mode to create an interface with a program simply use the following steps.

1. Select the buffer which you want to run the subprocess in. The value of the 'buffer-file-name' attribute of the buffer defines the working directory of the subprocess.
2. Set the variables described above to suitable values.
3. Call the 'shell-mode' function.
4. Reset the values of the 'mode-name' and 'major-mode' if necessary and install your own keymaps.

Remember that commands essential to the Shell mode (and hence your program) are contained in the two keymaps 'shell-keymap' and 'shell-ctrl-c-keymap'. If you need to bind your own commands to either of these prefixes make copies of these keymaps (using the function 'copy-sequence') and bind to the copies.

For example the gdb interface installs its own key bindings from the 'Ctrl-c' prefix by doing the following in its initialisation.

```
(defvar gdb-ctrl-c-keymap (copy-sequence shell-ctrl-c-keymap))
(bind-keys gdb-ctrl-c-keymap
 ;; Gdb mode 'Ctrl-c' prefix bindings follow
 ...)
```

- Function: shell-mode

This function installs the Shell mode and starts a subprocess running in the current buffer.

The variables 'shell-program', 'shell-program-args', 'shell-prompt-regexp', 'shell-callback-function' and 'shell-output-stream' control the program executed and how it will execute.

The process object created is stored in the buffer-local variable 'shell-process'.

- Variable: shell-process

This buffer-local variable contains the process object which the Shell mode started running in this buffer. If it is 'nil' no such process exists.

- Variable: shell-keymap

The root keymap of the Shell mode.

- Variable: shell-ctrl-c-keymap

The keymap containing the key bindings of the commands in Shell mode with a prefix of 'Ctrl-c'.

See the Lisp program 'gdb.jl' for an example of how to use the Shell mode as the user interface with an external program.

---

## 1.292 jade.guide/Miscellaneous Functions

### Miscellaneous Functions

=====

This section of the manual documents functions and features which don't comfortably fit elsewhere in this manual.

#### System Information

Getting details about the host

#### User Information

The name of the user

#### Environment Variables

Reading and writing the environment

#### System Time

Getting the current time

#### Revision Information

How to check Jade's revision numbers

## 1.293 jade.guide/System Information

### System Information

-----

- Function: x11-p  
This function returns 't' when Jade is running on the X11 window system.
- Function: unix-p  
This function returns 't' when Jade is running on a variant of the Unix operating system.
- Function: amiga-p  
This function returns 't' when Jade is running on an Amiga.
- Function: system-name  
This function returns a string naming the host that Jade is running on. When possible this will include the name of the domain as well.

In the Amiga version of Jade the environment variable 'HOSTNAME' is assumed to contain the host's name.

## 1.294 jade.guide/User Information

### User Information

-----

- Function: user-login-name

This function returns a string containing the login name of the user.

In the Amiga version this is taken from the environment variable 'USERNAME'.

```
(user-login-name)
=> "jsh"
```

- Function: user-real-name

This function returns a string containing the 'real' name of the user; the format of the string will depend on the host system.

In the Amiga version this is taken from the 'REALNAME' environment variable.

```
(user-real-name)
=> "John Harper"
```

- Function: user-home-directory

This function returns the name of the user's home directory terminated by a slash character ('/').

The first place this is looked for is in the 'HOME' environment variable; if this variable doesn't exist we either use the 'SYS:' logical device in AmigaDOS or consult the passwd file when in Unix.

```
(user-home-directory)
=> "/home/jsh/"
```

## 1.295 jade.guide/Environment Variables

### Environment Variables

-----

- Function: getenv VARIABLE-NAME

This function returns the value (a string) of the environment variable called VARIABLE-NAME. If the specified variable doesn't exist 'nil' is returned.

```
(getenv "OSTYPE")
=> "Linux"
```

- Function: setenv VARIABLE-NAME NEW-VALUE

This function sets the value of the environment variable called VARIABLE-NAME to NEW-VALUE. NEW-VALUE can either be a string containing the new contents of the variable or 'nil', in which

case the environment variable is deleted.

## 1.296 jade.guide/System Time

### System Time

-----

No matter what operating system Jade is running on it always an integer to store a time value. Generally this will be the number of seconds since some previous date.

The only thing a Lisp program is allowed to assume about a time value is that as time passes the time value *\*increases\**. This means that it's possible to compare two time values and know which is the newer.

- Function: `current-time`

Returns an integer denoting the current time.

```
(current-time)
=> 780935736
```

- Function: `current-time-string`

This function returns a string stating the current time and date in a fixed format. An example of the format is,

```
Fri Sep 30 15:20:56 1994
```

Each field will always be in the same place, for example,

```
Thu Sep 1 12:13:14 1994
```

```
(current-time-string)
=> "Fri Sep 30 15:20:56 1994"
```

## 1.297 jade.guide/Revision Information

### Revision Information

-----

- Function: `major-version-number`

This function returns a number defining the major version of the editor.

```
(major-version-number)
=> 3
```

- Function: `minor-version-number`

Returns a number defining the minor version of the editor.

---



```
(minor-version-number)
=> 2
```

## 1.298 jade.guide/Debugging

### Debugging

=====

When you have written a Lisp program you will have to debug it (unless all your programs work first time?). There are two main classes of errors; syntax errors and semantic errors.

Syntax errors occur when the text you've typed out to represent your program is not a valid representation of a Lisp object (since a program is simply an ordered set of Lisp objects). When you try to load your program the Lisp reader will find the syntax error and tell you about, unfortunately though it probably won't be able to tell you exactly where the error is.

The most common source of syntax errors is too few or too many parentheses; the 'Ctrl-Meta-f' and 'Ctrl-Meta-b' commands can be used to show the structure of the program as the Lisp reader sees it.

Semantic errors are what we normally call bugs -- errors in logic, the program is syntactically correct but doesn't do what you want it to. For these types of errors Jade provides a simple debugger which allows you to single step through the Lisp forms of your program as they are being evaluated.

There are several ways to enter the Lisp debugger; functions can be marked so that they cause the debugger to be entered when they are called, breakpoints can be written in functions or it can be called explicitly with a form to step through.

- Command: trace SYMBOL

This command marks the symbol SYMBOL so that each time the function stored in the function cell of SYMBOL is called the debugger is entered immediately.

When called interactively SYMBOL is prompted for.

- Command: untrace SYMBOL

The opposite of 'trace' -- unmarks the symbol.

- Function: break

This function causes the debugger to be entered immediately. By putting the form '(break)' at suitable points in your program simple breakpoints can be created.

- Command: step FORM

This function invokes the debugger to step through the form FORM.

When called interactively FORM is prompted for.

---

Whenever the Lisp debugger is entered the form waiting to be evaluated is printed at the bottom of the buffer, at this point the special debugger commands available are,

`'Ctrl-c Ctrl-s'`

Step into the current form; this means that in a list form the debugger is used to evaluate each argument in turn.

`'Ctrl-c Ctrl-i'`

Ignore the current form; makes the current form immediately return `'nil'`.

`'Ctrl-c Ctrl-n'`

Continue evaluating forms normally until the next form at the current level is entered, then re-enter the debugger.

`'Ctrl-c Ctrl-r'`

Continue execution normally. Note that this command is the one to use when an error has been trapped.

`'Ctrl-c Ctrl-b'`

Print a backtrace of the current Lisp call stack, note that calls of primitive functions aren't currently recorded in this stack.

`'Ctrl-c Ctrl-x'`

Prompt for a Lisp form, evaluate it and return this value as the result of the current form.

After the form has been evaluated (i.e. after you've typed one of the commands above) the value of the form is printed in the buffer, prefixed by the string `'=> '`.

Note that it is also possible to make certain types of errors invoke the debugger immediately they are signalled, see

Errors

.

## 1.299 jade.guide/Tips

Tips

====

This section of the manual gives advice about programming in Jade.

Obviously there is no *\*need\** to religiously follow every single one, but following these tips will make your programs easier to read and (hopefully) more efficient overall.

For advice on getting the most out of the compiler, see

Compilation Tips

.

---

|                |                                      |
|----------------|--------------------------------------|
| Comment Styles | Different types of comments          |
| Program Layout | How I lay out the programs I write   |
| General Tips   | Do's and Don't's of Jade programming |

## 1.300 jade.guide/Comment Styles

### Comment Styles

-----

As already described, single-line comments in Lisp are introduced by a semi-colon (`';`) character. By convention a different number of semi-colons is used to introduce different types of comments,

`';`

A comment referring to the line of Lisp code that it occurs on, comments of this type are usually indented to the same depth, on the right of the Lisp code. When editing in Lisp mode the command `'Meta-;` can be used to insert a comment of this type.

For example,

```
(defconst op-call 0x08) ;call (stk[n] stk[n-1] ... stk[0])
 ; pops n values, replacing the
 ; function with the result.
(defconst op-push 0x10) ;pushes constant # n
```

`;;`

Comments starting with two semi-colons are written on a line of their own and indented to the same depth as the next line of Lisp code. They describe the following lines of code.

For example,

```
;; Be sure to remove any partially written dst-file.
(let
 ((fname (concat file-name ?c)))
 (when (file-exists-p fname)
 (delete-file fname)))
```

Comments of this type are also placed before a function definition to describe the function. This saves wasting memory with a documentation string in a module's internal functions.

For example,

```
;; Compile a form which occurred at the 'top-level' into a
```

```
;; byte code form.
;; defuns, defmacros, defvars, etc... are treated specially.
;; require forms are evaluated before being output uncompiled;
;; this is so any macros are brought in before they're used.
(defun comp-compile-top-form (form)
 ...
```

`';;;'`

This type of comment always starts in the first column of the line, they are used to make general comments about a program and don't refer to any function or piece of code in particular.

For example,

```
;;; Notes:
;;;
;;; Instruction Encoding
;;; =====
;;; Instructions which get an argument (with opcodes of zero up to
...

```

`';;;;'`

Each program should have a comment of this type as its first line, the body of the comment is the name of the file, two dashes and a brief description of what the program does. They always start in the first column.

For example,

```
;;;; compiler.jl -- Simple compiler for Lisp files/forms
```

If you adhere to these standards the indentation functions provide by the Lisp mode will indent your comments to the correct depth.

## 1.301 jade.guide/Program Layout

Program Layout

-----

The layout that I have used for all the Lisp programs included with Jade is as follows, obviously this isn't ideal but it seems ok.

1. The first line of the file is the header comment, including the name of the file and its general function.
2. Copyright banner.
3. Any 'require' forms needed followed by a 'provide' form for this module. The 'require' forms should be before the 'provide' in case the required modules aren't available.
4. Variable and constant definitions. As a variable is defined any initialisation it needs is done immediately afterwards. For example a keymap is defined with 'defvar' then initialised with the

`'bind-keys'` function.

For example,

```
(defvar debug-buffer (make-buffer "*debugger*")
 "Buffer to use for the Lisp debugger.")
(set-buffer-special debug-buffer t)
(add-buffer debug-buffer)

(defvar debug-ctrl-c-keymap (make-keylist)
 "Keymap for debugger's ctrl-c prefix.")
(bind-keys debug-ctrl-c-keymap
 "Ctrl-s" 'debug-step
 ...)
```

5. Finally the functions which make up the program, it often improves readability if the entry points to the program are defined first.

## 1.302 jade.guide/General Tips

### General Tips

-----

The following are some general items of advice; you don't have to follow them but they are the result of experience!

- \* Jade only has one name-space for all the symbols ever created, this could lead to naming clashes if care isn't taken.

When you write a program all the symbols it creates should be prefixed by a name derived from the name of the program in some way. For example, in the program `'isearch.jl'` all functions and variable names are prefixed by the string `'isearch-'`, giving `'isearch-cancel'` and so on. Note that the prefix doesn't have to be the exact name of the file, the program `'buffer-menu.jl'` uses the prefix `'bm-'`.

The entry points to a module (i.e. the names of the commands it provides) should *not* have a prefix, simply give them a descriptive name (but try not to make it too long!).

Don't bother giving local variables these prefixes unless they are used by several functions in the program.

- \* Use the `'recursive-edit'` function as little as possible; it can be *very* confusing for the user! When at all possible use keymaps to create user interfaces.
- \* Use the Lisp mode to indent your programs; not only does it save a lot of time it also makes it easier for other people to read them.
- \* Errors should always be reported by either `'error'` or `'signal'`, don't just print a message or call `'beep'`.

- \* Don't redefine existing functions unless absolutely possible: try to use hooks. If there is no hook where you want one, mail me about it and I may put one in the next release.
- \* Don't compile your program until you're sure it works! The debugger only works properly with uncompiled code.
- \* Use constants sparingly: personally, I only use them where the constants are numeric.
- \* Remember to define macros before they are used, otherwise they won't be compiled inline. The same can happen if you don't 'require' a file that a macro is defined in before using the macro definition.
- \* As I said in the compilation tips (see  
Compilation Tips  
) , try to  
use iteration instead of recursion. Also the 'memq' and 'assq'  
types of functions can be used to search some types of list  
structures very quickly.
- \* When writing modes don't bind any unmodified keys to the prefix  
'Ctrl-c', these are reserved for customisation by users.

### 1.303 jade.guide/Reporting Bugs

#### Reporting Bugs

\*\*\*\*\*

If you think you've found a bug in Jade I want to know about it, there is a list of problems that I am aware of in the 'src/BUGS' file, if yours appears in there tell me anyway to make me fix it.

When submitting bug reports I need to know as much as possible, both about the problem and the circumstances in which it occurs. In general, send me as much information as possible, even if you think it's probably irrelevant.

If you can, contact me via email, my address is 'jsh@ukc.ac.uk'. If you don't get a reply within about a week it's probably a university vacation -- this means that I won't get your message for a while; if it's important try my postal address, this is,

John Harper  
91 Springdale Road  
Broadstone  
Dorset  
BH18 9BW  
England

As well as bugs I'm interested in any comments you have about the editor, even if you just tell me you hate it (as long as you say \*why\*

---

you hate it!).

## 1.304 jade.guide/Function Index

### Function Index

\*\*\*\*\*

|                      |                       |
|----------------------|-----------------------|
| %                    | Arithmetic Functions  |
| *                    | Arithmetic Functions  |
| +                    | Arithmetic Functions  |
| -                    | Arithmetic Functions  |
| /                    | Arithmetic Functions  |
| /=                   | Numeric Predicates    |
| 1+                   | Arithmetic Functions  |
| 1-                   | Arithmetic Functions  |
| <                    | Comparison Predicates |
| <=                   | Comparison Predicates |
| =                    | Numeric Predicates    |
| >                    | Comparison Predicates |
| >=                   | Comparison Predicates |
| abort-recursive-edit | Recursive Edits       |
| add-buffer           | The Buffer List       |

---

|                        |                        |
|------------------------|------------------------|
| add-hook               | Normal Hooks           |
| add-minor-mode         | Writing Minor Modes    |
| alpha-char-p           | Characters             |
| alphanumericp          | Characters             |
| amiga-p                | System Information     |
| and                    | Conditional Structures |
| append                 | Building Lists         |
| apply                  | Calling Functions      |
| apropos                | Obarrays               |
| aref                   | Array Functions        |
| arrayp                 | Array Functions        |
| aset                   | Array Functions        |
| ash                    | Bitwise Functions      |
| asm-mode               | Asm mode               |
| assoc                  | Association Lists      |
| assq                   | Association Lists      |
| atom                   | Cons Cells             |
| auto-save-file-newer-p | Controlling Auto-Saves |
| auto-save-function     | Controlling Auto-Saves |

---



---

|                     |                     |
|---------------------|---------------------|
| auto-save-mode      | Auto-Save mode      |
| autoload            | Autoloading         |
| backspace-char      | Deletion Functions  |
| backup-file         | Making Backups      |
| backward-exp        | Expression Movement |
| backward-kill-exp   | Kill Functions      |
| backward-word       | Word Movement       |
| backwards-kill-word | Kill Functions      |
| beep                | Displaying Messages |
| bind-keys           | Binding Keys        |
| block-end           | Block Marking       |
| block-kill          | Block Marking       |
| block-start         | Block Marking       |
| block-toggle        | Block Marking       |
| blockp              | Block Marking       |
| boundp              | Void Variables      |
| break               | Debugging           |
| buffer-changes      | Buffer Attributes   |
| buffer-end          | Buffer Extremes     |

---

---

|                    |                          |
|--------------------|--------------------------|
| buffer-file-name   | Buffer Attributes        |
| buffer-glyph-table | Buffer Glyph Tables      |
| buffer-modified-p  | Modifications to Buffers |
| buffer-name        | Buffer Attributes        |
| buffer-read-only-p | Read-Only Buffers        |
| buffer-special-p   | Special Buffers          |
| buffer-start       | Buffer Extremes          |
| buffer-variables   | Buffer Attributes        |
| bufferp            | Buffers                  |
| bury-buffer        | The Buffer List          |
| c-mode             | C mode                   |
| call-command       | Calling Commands         |
| capitalize-word    | Translation Functions    |
| car                | Cons Cells               |
| catch              | Catch and Throw          |
| cdr                | Cons Cells               |
| centre-display     | Rendering                |
| char-downcase      | Characters               |
| char-to-glyph-pos  | Glyph Positions          |

---

---

|                     |                         |
|---------------------|-------------------------|
| char-upcase         | Characters              |
| clear-buffer        | Buffer Contents         |
| close               | Destroying File Objects |
| close-other-windows | Closing Windows         |
| close-window        | Closing Windows         |
| commandp            | Calling Commands        |
| compile-directory   | Compilation Functions   |
| compile-file        | Compilation Functions   |
| compile-form        | Compilation Functions   |
| compile-lisp-lib    | Compilation Functions   |
| concat              | Strings                 |
| cond                | Conditional Structures  |
| cons                | Cons Cells              |
| consp               | Cons Cells              |
| const-variable-p    | Constant Variables      |
| continue-process    | Process States          |
| copy-area           | Buffer Contents         |
| copy-area-as-kill   | Kill Functions          |
| copy-block          | Buffer Contents         |

---

---

|                      |                        |
|----------------------|------------------------|
| copy-block-as-kill   | Kill Functions         |
| copy-file            | Manipulating Files     |
| copy-pos             | Positions              |
| copy-rect            | Rectangular Editing    |
| copy-sequence        | Sequence Functions     |
| copy-stream          | Output Functions       |
| current-buffer       | The Current Buffer     |
| current-event        | Event Loop Info        |
| current-event-string | Event Loop Info        |
| current-time         | System Time            |
| current-time-string  | System Time            |
| current-window       | The Current Window     |
| cursor               | Rendering              |
| cursor-pos           | The Cursor Position    |
| cut-area             | Deletion Functions     |
| cut-block            | Deletion Functions     |
| cut-rect             | Rectangular Editing    |
| default-boundp       | Buffer-Local Variables |
| default-glyph-table  | Buffer Glyph Tables    |

---

---

|                       |                        |
|-----------------------|------------------------|
| default-value         | Buffer-Local Variables |
| defconst              | Defining Variables     |
| defmacro              | Defining Macros        |
| defun                 | Defining Functions     |
| defvar                | Defining Variables     |
| delete                | Modifying Lists        |
| delete-area           | Deletion Functions     |
| delete-auto-save-file | Controlling Auto-Saves |
| delete-block          | Deletion Functions     |
| delete-char           | Deletion Functions     |
| delete-file           | Manipulating Files     |
| delete-if             | Mapping Functions      |
| delete-if-not         | Mapping Functions      |
| delete-rect           | Rectangular Editing    |
| delq                  | Modifying Lists        |
| destroy-buffer        | Destroying Buffers     |
| destroy-window        | Closing Windows        |
| digit-char-p          | Characters             |
| directory-files       | Reading Directories    |

---

---

|                  |                           |
|------------------|---------------------------|
| disassemble-fun  | Disassembly               |
| downcase-area    | Translation Functions     |
| downcase-word    | Translation Functions     |
| elt              | Sequence Functions        |
| empty-line-p     | Misc Text Functions       |
| eq               | Equality Predicates       |
| eq1              | Equality Predicates       |
| equal            | Equality Predicates       |
| error-protect    | Errors                    |
| eval             | Evaluation                |
| eval-hook        | Normal Hooks              |
| event-name       | Input Events              |
| eventp           | Input Events              |
| expand-file-name | File Names                |
| fboundp          | Named Functions           |
| file-binding     | Functions on File Objects |
| file-bound-p     | File Object Predicates    |
| file-directory-p | File Information          |
| file-eof-p       | File Object Predicates    |

---

---

|                        |                            |
|------------------------|----------------------------|
| file-exists-p          | File Information           |
| file-modes             | File Information           |
| file-modtime           | File Information           |
| file-name-concat       | File Names                 |
| file-name-directory    | File Names                 |
| file-name-nondirectory | File Names                 |
| file-newer-than-file-p | File Information           |
| file-nlinks            | File Information           |
| file-owner-p           | File Information           |
| file-readable-p        | File Information           |
| file-regular-p         | File Information           |
| file-symlink-p         | File Information           |
| file-writable-p        | File Information           |
| filep                  | File Object Predicates     |
| fill-mode              | Fill mode                  |
| find-alternate-file    | Reading Files Into Buffers |
| find-comment-pos       | Mode-Specific Comments     |
| find-file              | Reading Files Into Buffers |
| find-file-read-only    | Reading Files Into Buffers |

---

---

|                  |                           |
|------------------|---------------------------|
| find-next-char   | Searching Buffers         |
| find-next-regexp | Searching Buffers         |
| find-next-string | Searching Buffers         |
| find-prev-char   | Searching Buffers         |
| find-prev-regexp | Searching Buffers         |
| find-prev-string | Searching Buffers         |
| find-symbol      | Oarrays                   |
| flush-file       | Functions on File Objects |
| flush-output     | Rendering                 |
| fmakunbound      | Named Functions           |
| font-name        | Window Font               |
| font-x-size      | Window Font               |
| font-y-size      | Window Font               |
| format           | Output Functions          |
| forward-exp      | Expression Movement       |
| forward-word     | Word Movement             |
| fset             | Named Functions           |
| funcall          | Calling Functions         |
| function         | Anonymous Functions       |

---



---

|                          |                       |
|--------------------------|-----------------------|
| functionp                | Functions             |
| garbage-collect          | Garbage Collection    |
| gensym                   | Creating Symbols      |
| get                      | Property Lists        |
| get-buffer               | Buffer Attributes     |
| get-char                 | Buffer Contents       |
| get-file-buffer          | Buffer Attributes     |
| get-glyph                | Glyph Table Basics    |
| get-output-stream-string | Output Streams        |
| getenv                   | Environment Variables |
| glyph-table-p            | Glyph Tables          |
| glyph-to-char-pos        | Glyph Positions       |
| goto-buffer              | The Current Buffer    |
| goto-buffer-end          | Buffer Extremes       |
| goto-buffer-start        | Buffer Extremes       |
| goto-char                | The Cursor Position   |
| goto-left-char           | Character Movement    |
| goto-mark                | Moving to Marks       |
| goto-next-char           | Character Movement    |

---

---

|                    |                            |
|--------------------|----------------------------|
| goto-next-line     | Line Movement              |
| goto-next-tab      | Tab Movement               |
| goto-prev-char     | Character Movement         |
| goto-prev-line     | Line Movement              |
| goto-prev-tab      | Tab Movement               |
| goto-right-char    | Character Movement         |
| if                 | Conditional Structures     |
| in-word-p          | Word Movement              |
| indent-area        | Indentation Functions      |
| indent-line        | Indentation Functions      |
| indent-pos         | Indentation Functions      |
| indent-to          | Indentation Functions      |
| indented-text-mode | Indented-Text mode         |
| init-mode          | Installing Modes           |
| insert             | Insertion Functions        |
| insert-block       | Insertion Functions        |
| insert-file        | Reading Files Into Buffers |
| insert-rect        | Rectangular Editing        |
| integerp           | Numbers                    |

---

---

|                          |                         |
|--------------------------|-------------------------|
| intern                   | Interning               |
| intern-symbol            | Interning               |
| interrupt-process        | Signalling Processes    |
| jade-byte-code           | Compilation Functions   |
| keymapp                  | Keymaps                 |
| kill-all-local-variables | Buffer-Local Variables  |
| kill-area                | Kill Functions          |
| kill-block               | Kill Functions          |
| kill-buffer              | Destroying Buffers      |
| kill-exp                 | Kill Functions          |
| kill-line                | Kill Functions          |
| kill-local-variable      | Buffer-Local Variables  |
| kill-mode                | Installing Modes        |
| kill-process             | Signalling Processes    |
| kill-string              | Kill Functions          |
| kill-whole-line          | Kill Functions          |
| kill-word                | Kill Functions          |
| killed-string            | Kill Functions          |
| last                     | Accessing List Elements |

---

---

|                      |                    |
|----------------------|--------------------|
| last-event           | Event Loop Info    |
| latin-1-mode         | Latin-1 mode       |
| left-char            | Character Movement |
| length               | Sequence Functions |
| let                  | Local Variables    |
| let*                 | Local Variables    |
| lisp-mode            | Lisp mode          |
| list                 | Building Lists     |
| listp                | Lists              |
| load                 | Load Function      |
| logand               | Bitwise Functions  |
| logior               | Bitwise Functions  |
| lognot               | Bitwise Functions  |
| logxor               | Bitwise Functions  |
| looking-at           | String Matching    |
| lookup-event         | Input Events       |
| lookup-event-binding | Key Lookup         |
| lower-case-p         | Characters         |
| lsh                  | Bitwise Functions  |

---

---

|                            |                        |
|----------------------------|------------------------|
| macroexpand                | Macro Expansion        |
| major-version-number       | Revision Information   |
| make-auto-save-name        | Controlling Auto-Saves |
| make-buffer                | Creating Buffers       |
| make-buffer-name           | Buffer Attributes      |
| make-glyph-table           | Creating Glyph Tables  |
| make-keylist               | Creating Keymaps       |
| make-keytab                | Creating Keymaps       |
| make-list                  | Building Lists         |
| make-local-variable        | Buffer-Local Variables |
| make-mark                  | Creating Marks         |
| make-obarray               | Obarrays               |
| make-process               | Process Objects        |
| make-string                | Strings                |
| make-string-input-stream   | Input Streams          |
| make-string-output-stream  | Output Streams         |
| make-symbol                | Creating Symbols       |
| make-variable-buffer-local | Buffer-Local Variables |
| make-vector                | Vectors                |

---

---

|                      |                         |
|----------------------|-------------------------|
| make-window          | Opening Windows         |
| makunbound           | Void Variables          |
| mapc                 | Mapping Functions       |
| mapcar               | Mapping Functions       |
| mark-block           | Block Marking           |
| mark-file            | Mark Components         |
| mark-pos             | Mark Components         |
| mark-resident-p      | Mark Residency          |
| mark-whole-buffer    | Block Marking           |
| mark-word            | Block Marking           |
| markp                | Marks                   |
| match-end            | Regexp Functions        |
| match-start          | Regexp Functions        |
| member               | Accessing List Elements |
| memq                 | Accessing List Elements |
| message              | Displaying Messages     |
| minor-version-number | Revision Information    |
| nconc                | Modifying Lists         |
| newline-and-indent   | Indentation Functions   |

---

---

|                |                            |
|----------------|----------------------------|
| next-char      | Character Movement         |
| next-line      | Line Movement              |
| next-screen    | Rendering                  |
| next-tab       | Tab Movement               |
| not            | Conditional Structures     |
| nreverse       | Modifying Lists            |
| nth            | Accessing List Elements    |
| nthcdr         | Accessing List Elements    |
| numberp        | Numbers                    |
| offset-to-pos  | Positions and Offsets      |
| open           | Creating File Objects      |
| open-buffer    | Creating Buffers           |
| open-file      | Reading Files Into Buffers |
| open-line      | Insertion Functions        |
| open-window    | Opening Windows            |
| or             | Conditional Structures     |
| overwrite-mode | Overwrite mode             |
| pos            | Positions                  |
| pos-col        | Position Components        |

---

---

|                         |                        |
|-------------------------|------------------------|
| pos-line                | Position Components    |
| pos-to-offset           | Positions and Offsets  |
| posp                    | Positions              |
| prefix-numeric-argument | Prefix Arguments       |
| prev-char               | Character Movement     |
| prev-line               | Line Movement          |
| prev-screen             | Rendering              |
| prev-tab                | Tab Movement           |
| prinl                   | Output Functions       |
| prinl-to-string         | Output Functions       |
| princ                   | Output Functions       |
| print                   | Output Functions       |
| process-args            | Process Objects        |
| process-connection-type | Asynchronous Processes |
| process-dir             | Process Objects        |
| process-exit-status     | Process Information    |
| process-exit-value      | Process Information    |
| process-function        | Process States         |
| process-id              | Process Information    |

---



---

|                       |                       |
|-----------------------|-----------------------|
| process-in-use-p      | Process States        |
| process-output-stream | Process I-O           |
| process-prog          | Process Objects       |
| process-running-p     | Process States        |
| process-stopped-p     | Process States        |
| processp              | Process Objects       |
| prog1                 | Sequencing Structures |
| prog2                 | Sequencing Structures |
| progn                 | Sequencing Structures |
| prompt-for-buffer     | Prompting             |
| prompt-for-command    | Prompting             |
| prompt-for-directory  | Prompting             |
| prompt-for-file       | Prompting             |
| prompt-for-function   | Prompting             |
| prompt-for-lisp       | Prompting             |
| prompt-for-number     | Prompting             |
| prompt-for-string     | Prompting             |
| prompt-for-symbol     | Prompting             |
| prompt-for-variable   | Prompting             |

---

---

|                  |                            |
|------------------|----------------------------|
| prompt-from-list | Prompting                  |
| provide          | Features                   |
| put              | Property Lists             |
| quote            | Quoting                    |
| rassoc           | Association Lists          |
| rassq            | Association Lists          |
| read             | Input Functions            |
| read-buffer      | Reading Files Into Buffers |
| read-char        | Input Functions            |
| read-event       | Reading Events             |
| read-file        | Reading and Writing Files  |
| read-file-until  | Functions on File Objects  |
| read-from-string | Input Functions            |
| read-line        | Input Functions            |
| recover-file     | Controlling Auto-Saves     |
| rect-blocks-p    | Block Marking              |
| recursion-depth  | Recursive Edits            |
| recursive-edit   | Recursive Edits            |
| refresh-all      | Rendering                  |

---

---

|                        |                            |
|------------------------|----------------------------|
| regexp-expand          | String Matching            |
| regexp-expand-line     | String Matching            |
| regexp-match           | String Matching            |
| regexp-match-line      | String Matching            |
| regexp-quote           | Regex Functions            |
| remove-buffer          | The Buffer List            |
| remove-hook            | Normal Hooks               |
| remove-minor-mode      | Writing Minor Modes        |
| rename-file            | Manipulating Files         |
| replace-regexp         | Replacing Strings          |
| replace-string         | Replacing Strings          |
| require                | Features                   |
| return                 | Function Exits             |
| reverse                | Building Lists             |
| revert-buffer          | Reading Files Into Buffers |
| right-char             | Character Movement         |
| rotate-buffers-forward | The Buffer List            |
| rplaca                 | Cons Cells                 |
| rplacd                 | Cons Cells                 |

---

---

|                        |                          |
|------------------------|--------------------------|
| run-process            | Synchronous Processes    |
| save-and-quit          | Writing Buffers          |
| save-file              | Writing Buffers          |
| save-file-as           | Writing Buffers          |
| save-some-buffers      | Writing Buffers          |
| screen-bottom-line     | Window Information       |
| screen-first-column    | Window Information       |
| screen-height          | Window Information       |
| screen-last-column     | Window Information       |
| screen-top-line        | Window Information       |
| screen-width           | Window Information       |
| sequencep              | Sequences                |
| set                    | Setting Variables        |
| set-buffer-file-name   | Buffer Attributes        |
| set-buffer-glyph-table | Buffer Glyph Tables      |
| set-buffer-modified    | Modifications to Buffers |
| set-buffer-name        | Buffer Attributes        |
| set-buffer-read-only   | Read-Only Buffers        |
| set-buffer-special     | Special Buffers          |

---

---

|                             |                        |
|-----------------------------|------------------------|
| set-char                    | Buffer Contents        |
| set-const-variable          | Constant Variables     |
| set-current-buffer          | The Current Buffer     |
| set-current-window          | The Current Window     |
| set-default                 | Buffer-Local Variables |
| set-file-modes              | File Information       |
| set-font                    | Window Font            |
| set-glyph                   | Glyph Table Basics     |
| set-indent-pos              | Indentation Functions  |
| set-mark                    | Altering Marks         |
| set-pos-col                 | Position Components    |
| set-pos-line                | Position Components    |
| set-process-args            | Process Objects        |
| set-process-connection-type | Asynchronous Processes |
| set-process-directory       | Process Objects        |
| set-process-function        | Process States         |
| set-process-output-stream   | Process I-O            |
| set-process-prog            | Process Objects        |
| set-rect-blocks             | Block Marking          |

---

---

|                 |                        |
|-----------------|------------------------|
| setenv          | Environment Variables  |
| setplist        | Property Lists         |
| setq            | Setting Variables      |
| setq-default    | Buffer-Local Variables |
| shell-mode      | Interactive Processes  |
| signal          | Errors                 |
| signal-process  | Signalling Processes   |
| sleep-window    | Iconifying Windows     |
| space-char-p    | Characters             |
| split-line      | Insertion Functions    |
| start-process   | Asynchronous Processes |
| step            | Debugging              |
| stop-process    | Process States         |
| streamp         | Streams                |
| string<         | Strings                |
| string=         | Strings                |
| stringp         | Strings                |
| substring       | Strings                |
| symbol-function | Named Functions        |

---

---

|                    |                       |
|--------------------|-----------------------|
| symbol-name        | Symbol Attributes     |
| symbol-plist       | Property Lists        |
| symbol-value       | Variables             |
| symbolp            | Symbols               |
| system-name        | System Information    |
| tab-with-spaces    | Indentation Functions |
| texinfo-mode       | Texinfo mode          |
| text-mode          | Text mode             |
| throw              | Catch and Throw       |
| tmp-file-name      | File Names            |
| toggle-iconic      | Iconifying Windows    |
| toggle-rect-blocks | Block Marking         |
| top-level          | Recursive Edits       |
| trace              | Debugging             |
| translate-area     | Translation Functions |
| translate-string   | Translation Functions |
| transpose-chars    | Transpose Functions   |
| transpose-exps     | Transpose Functions   |
| transpose-items    | Transpose Functions   |

---

---

|                     |                        |
|---------------------|------------------------|
| transpose-words     | Transpose Functions    |
| unbind-keys         | Binding Keys           |
| undo                | Controlling Undo       |
| unintern            | Interning              |
| unix-p              | System Information     |
| unless              | Conditional Structures |
| unsleep-window      | Iconifying Windows     |
| untrace             | Debugging              |
| unwind-protect      | Cleanup Forms          |
| upcase-area         | Translation Functions  |
| upcase-word         | Translation Functions  |
| upper-case-p        | Characters             |
| useless-function    | Descriptions           |
| user-home-directory | User Information       |
| user-login-name     | User Information       |
| user-real-name      | User Information       |
| vector              | Vectors                |
| vectorp             | Vectors                |
| when                | Conditional Structures |

---



---

|                   |                           |
|-------------------|---------------------------|
| while             | Looping Structures        |
| window-asleep-p   | Iconifying Windows        |
| window-bar-height | Window Information        |
| window-count      | Window Information        |
| window-height     | Window Information        |
| window-id         | Window Information        |
| window-left-edge  | Window Information        |
| window-top-edge   | Window Information        |
| window-width      | Window Information        |
| windowp           | Windows                   |
| with-buffer       | The Current Buffer        |
| with-window       | The Current Window        |
| word-start        | Word Movement             |
| write             | Output Functions          |
| write-buffer      | Writing Buffers           |
| write-buffer-area | Writing Buffers           |
| write-file        | Reading and Writing Files |
| write-file        | Writing Buffers           |
| x11-p             | System Information        |

---

|                |                     |
|----------------|---------------------|
| y-or-n-p       | Prompting           |
| yank           | Insertion Functions |
| yank-rectangle | Rectangular Editing |
| yank-to-mouse  | Insertion Functions |
| yes-or-no-p    | Prompting           |
| zerop          | Numeric Predicates  |

### 1.305 jade.guide/Variable Index

#### Variable Index

\*\*\*\*\*

|                     |                        |
|---------------------|------------------------|
| asm-comment         | Asm mode               |
| asm-indent          | Asm mode               |
| asm-mode-hook       | Asm mode               |
| auto-save-hook      | Controlling Auto-Saves |
| auto-save-interval  | Auto-Saving Files      |
| auto-save-p         | Auto-Saving Files      |
| backup-by-copying   | Backup Files           |
| buffer-file-modtime | Buffer Date Stamps     |
| buffer-list         | The Buffer List        |

---

---

|                            |                        |
|----------------------------|------------------------|
| buffer-record-undo         | Controlling Undo       |
| buffer-undo-list           | Controlling Undo       |
| c-mode-auto-indent         | C mode                 |
| c-mode-hook                | C mode                 |
| c-mode-tab                 | C mode                 |
| comment-column             | Mode-Specific Comments |
| compile-error-expand       | Finding Errors         |
| compile-error-regexp       | Finding Errors         |
| compile-file-expand        | Finding Errors         |
| compile-line-expand        | Finding Errors         |
| ctrl-c-keymap              | Writing Major Modes    |
| ctrl-x-4-keymap            | Standard Keymaps       |
| ctrl-x-5-keymap            | Standard Keymaps       |
| ctrl-x-keymap              | Standard Keymaps       |
| current-prefix-arg         | Prefix Arguments       |
| debug-on-error             | Errors                 |
| default-auto-save-interval | Auto-Saving Files      |
| destroy-window-hook        | Closing Windows        |
| downcase-table             | Translation Functions  |

---

---

|                         |                            |
|-------------------------|----------------------------|
| else-backup-by-copying  | Backup Files               |
| enable-local-eval       | File Variables             |
| enable-local-variables  | File Variables             |
| features                | Features                   |
| fill-column             | Fill mode                  |
| garbage-threshold       | Garbage Collection         |
| global-keymap           | Standard Keymaps           |
| grains-of-sand          | Descriptions               |
| idle-garbage-threshold  | Idle Actions               |
| indented-text-mode-hook | Indented-Text mode         |
| inhibit-read-only       | Read-Only Buffers          |
| insert-file-hook        | Reading Files Into Buffers |
| keymap-path             | Key Lookup                 |
| kill-buffer-hook        | Destroying Buffers         |
| last-command            | Event Loop Info            |
| last-save-changes       | Buffer Modification Counts |
| last-save-time          | Buffer Modification Counts |
| last-user-save-changes  | Buffer Modification Counts |
| lisp-body-indent        | Lisp mode                  |

---

---

|                                    |                           |
|------------------------------------|---------------------------|
| <code>lisp-lib-dir</code>          | Load Function             |
| <code>lisp-mode-hook</code>        | Lisp mode                 |
| <code>load-path</code>             | Load Function             |
| <code>local-variable-lines</code>  | File Variables            |
| <code>major-mode</code>            | Writing Major Modes       |
| <code>major-mode-kill</code>       | Writing Major Modes       |
| <code>make-backup-files</code>     | Backup Files              |
| <code>make-window-hook</code>      | Opening Windows           |
| <code>max-scroll</code>            | Rendering                 |
| <code>max-undo-size</code>         | Undo                      |
| <code>meta-sends-esc</code>        | Modifiers                 |
| <code>mildly-special-buffer</code> | Special Buffers           |
| <code>minor-mode-keymap</code>     | Writing Minor Modes       |
| <code>minor-mode-list</code>       | Writing Minor Modes       |
| <code>minor-mode-names</code>      | Writing Minor Modes       |
| <code>mode-alist</code>            | Installing Modes          |
| <code>mode-backward-exp</code>     | Mode-Specific Expressions |
| <code>mode-comment-fun</code>      | Mode-Specific Comments    |
| <code>mode-forward-exp</code>      | Mode-Specific Expressions |

---

---

|                         |                            |
|-------------------------|----------------------------|
| mode-indent-line        | Mode-Specific Indentation  |
| mode-name               | Writing Major Modes        |
| next-keymap-path        | Key Lookup                 |
| obarray                 | Oarrays                    |
| prefix-arg              | Prefix Arguments           |
| pub-screen              | Opening Windows            |
| read-file-hook          | Reading Files Into Buffers |
| shell-callback-function | Interactive Processes      |
| shell-ctrl-c-keymap     | Interactive Processes      |
| shell-file-name         | Shell                      |
| shell-keymap            | Interactive Processes      |
| shell-mode-hook         | Shell                      |
| shell-output-stream     | Interactive Processes      |
| shell-process           | Interactive Processes      |
| shell-program           | Interactive Processes      |
| shell-program-args      | Interactive Processes      |
| shell-prompt-regexp     | Interactive Processes      |
| shell-prompt-regexp     | Shell                      |
| shell-whole-line        | Shell                      |

---

---

|                     |                       |
|---------------------|-----------------------|
| standard-input      | Input Streams         |
| standard-output     | Output Streams        |
| status-line-cursor  | Displaying Messages   |
| tab-size            | Buffer Attributes     |
| texinfo-mode-hook   | Texinfo mode          |
| text-mode-hook      | Text mode             |
| this-command        | Event Loop Info       |
| unbound-key-hook    | Event Loop Actions    |
| upcase-table        | Translation Functions |
| user-keymap         | Standard Keymaps      |
| window-closed-hook  | Event Loop Actions    |
| window-list         | Windows               |
| word-not-regexp     | Word Movement         |
| word-regexp         | Word Movement         |
| write-file-hook     | Writing Buffers       |
| x-scroll-step-ratio | Rendering             |
| y-scroll-step-ratio | Rendering             |

---

## 1.306 jade.guide/Key Index

### Key Index

\*\*\*\*\*

|           |                    |
|-----------|--------------------|
| !         | Query Replace      |
| %         | The Buffer Menu    |
| '         | Query Replace      |
| -         | The Buffer Menu    |
| .         | Asm mode           |
| .         | Query Replace      |
| 1         | Info Mode          |
| 1         | The Buffer Menu    |
| :         | Asm mode           |
| :         | C mode             |
| ?         | Info Mode          |
| Backspace | Info Mode          |
| Backspace | Query Replace      |
| Backspace | Incremental Search |
| Backspace | Editing Characters |
| DEL       | Editing Characters |

---



---

|            |                    |
|------------|--------------------|
| Down       | Editing Lines      |
| ESC        | Query Replace      |
| ESC        | Incremental Search |
| F1         | Using Marks        |
| F2         | Using Marks        |
| F3         | Using Marks        |
| HELP       | The Help System    |
| HELP a     | The Help System    |
| HELP b     | The Help System    |
| HELP e     | The Help System    |
| HELP f     | The Help System    |
| HELP h     | The Help System    |
| HELP i     | The Help System    |
| HELP k     | The Help System    |
| HELP m     | The Help System    |
| HELP v     | The Help System    |
| Left       | Editing Characters |
| LMB-CLICK2 | The Buffer Prompt  |
| RET        | Shell              |

---

---

|            |                       |
|------------|-----------------------|
| RET        | The Buffer Prompt     |
| RET        | The Buffer Menu       |
| RET        | Asm mode              |
| RET        | Query Replace         |
| RET        | Incremental Search    |
| RET        | Editing Buffers       |
| Right      | Editing Characters    |
| RMB-CLICK1 | The Buffer Prompt     |
| SPC        | Info Mode             |
| SPC        | Query Replace         |
| TAB        | The Buffer Prompt     |
| TAB        | The Buffer Menu       |
| TAB        | Texinfo mode          |
| TAB        | Lisp mode             |
| TAB        | C mode                |
| TAB        | Moving Around Buffers |
| Up         | Editing Lines         |
| {          | C mode                |
| }          | C mode                |

---

---

|                      |                       |
|----------------------|-----------------------|
| b                    | Info Mode             |
| Ctrl-@               | Using Marks           |
| Ctrl-DEL             | Editing Lines         |
| Ctrl-Down            | Moving Around Buffers |
| Ctrl-TAB             | Moving Around Buffers |
| Ctrl-Up              | Moving Around Buffers |
| Ctrl-a               | Shell                 |
| Ctrl-a               | Editing Lines         |
| Ctrl-b               | The Buffer Menu       |
| Ctrl-b               | Editing Characters    |
| Ctrl-c Ctrl-<        | Debugging Programs    |
| Ctrl-c Ctrl->        | Debugging Programs    |
| Ctrl-c Ctrl-b        | Debugging             |
| Ctrl-c Ctrl-b        | Debugging Programs    |
| Ctrl-c Ctrl-c        | Shell                 |
| Ctrl-c Ctrl-c {      | Texinfo mode          |
| Ctrl-c Ctrl-c }      | Texinfo mode          |
| Ctrl-c Ctrl-c c      | Texinfo mode          |
| Ctrl-c Ctrl-c Ctrl-m | Texinfo mode          |

---

---

|                 |                    |
|-----------------|--------------------|
| Ctrl-c Ctrl-c d | Texinfo mode       |
| Ctrl-c Ctrl-c e | Texinfo mode       |
| Ctrl-c Ctrl-c f | Texinfo mode       |
| Ctrl-c Ctrl-c i | Texinfo mode       |
| Ctrl-c Ctrl-c l | Texinfo mode       |
| Ctrl-c Ctrl-c m | Texinfo mode       |
| Ctrl-c Ctrl-c n | Texinfo mode       |
| Ctrl-c Ctrl-c s | Texinfo mode       |
| Ctrl-c Ctrl-c v | Texinfo mode       |
| Ctrl-c Ctrl-c ] | Texinfo mode       |
| Ctrl-c Ctrl-d   | Shell              |
| Ctrl-c Ctrl-d   | Debugging Programs |
| Ctrl-c Ctrl-f   | Debugging Programs |
| Ctrl-c Ctrl-i   | Debugging          |
| Ctrl-c Ctrl-l   | Debugging Programs |
| Ctrl-c Ctrl-n   | Debugging          |
| Ctrl-c Ctrl-n   | Shell              |
| Ctrl-c Ctrl-n   | Debugging Programs |
| Ctrl-c Ctrl-p   | Shell              |

---

---

|               |                    |
|---------------|--------------------|
| Ctrl-c Ctrl-r | Debugging          |
| Ctrl-c Ctrl-r | Debugging Programs |
| Ctrl-c Ctrl-s | Debugging          |
| Ctrl-c Ctrl-s | Debugging Programs |
| Ctrl-c Ctrl-t | Debugging Programs |
| Ctrl-c Ctrl-x | Debugging          |
| Ctrl-c Ctrl-z | Shell              |
| Ctrl-c Ctrl-\ | Shell              |
| Ctrl-d        | Shell              |
| Ctrl-d        | Editing Characters |
| Ctrl-e        | Editing Lines      |
| Ctrl-f        | The Buffer Menu    |
| Ctrl-f        | Editing Characters |
| Ctrl-g        | The Buffer Prompt  |
| Ctrl-g        | Incremental Search |
| Ctrl-h        | The Help System    |
| Ctrl-h a      | The Help System    |
| Ctrl-h b      | The Help System    |
| Ctrl-h e      | The Help System    |

---

---

|             |                     |
|-------------|---------------------|
| Ctrl-h f    | The Help System     |
| Ctrl-h h    | The Help System     |
| Ctrl-h i    | The Help System     |
| Ctrl-h k    | The Help System     |
| Ctrl-h m    | The Help System     |
| Ctrl-h v    | The Help System     |
| Ctrl-i      | Commands on Blocks  |
| Ctrl-j      | Lisp mode           |
| Ctrl-k      | Killing             |
| Ctrl-k      | Editing Lines       |
| Ctrl-k      | Editing Buffers     |
| Ctrl-l      | The Buffer Menu     |
| Ctrl-M      | Rectangular Blocks  |
| Ctrl-m      | Marking Blocks      |
| Ctrl-Meta-b | Editing Expressions |
| Ctrl-Meta-c | Recursive Editing   |
| Ctrl-Meta-f | Editing Expressions |
| Ctrl-Meta-k | Killing             |
| Ctrl-Meta-k | Editing Expressions |

---

---

|             |                       |
|-------------|-----------------------|
| Ctrl-Meta-t | Editing Expressions   |
| Ctrl-Meta-x | Lisp mode             |
| Ctrl-Meta-\ | Lisp mode             |
| Ctrl-Meta-\ | C mode                |
| Ctrl-n      | Editing Lines         |
| Ctrl-o      | Editing Lines         |
| Ctrl-p      | Editing Lines         |
| Ctrl-q      | Incremental Search    |
| Ctrl-r      | Query Replace         |
| Ctrl-r      | Incremental Search    |
| Ctrl-s      | The Buffer Menu       |
| Ctrl-s      | Incremental Search    |
| Ctrl-SPC    | Marking Blocks        |
| Ctrl-t      | Editing Characters    |
| Ctrl-u      | Command Arguments     |
| Ctrl-v      | Moving Around Buffers |
| Ctrl-w      | Query Replace         |
| Ctrl-w      | Incremental Search    |
| Ctrl-w      | Killing               |

---

---

|                 |                    |
|-----------------|--------------------|
| Ctrl-W          | Commands on Blocks |
| Ctrl-w          | Commands on Blocks |
| Ctrl-x #        | Client Editing     |
| Ctrl-x 0        | Killing Windows    |
| Ctrl-x 1        | Killing Windows    |
| Ctrl-x 2        | Creating Windows   |
| Ctrl-x 4 a      | Creating Windows   |
| Ctrl-x 4 b      | Creating Windows   |
| Ctrl-x 4 b      | Displaying Buffers |
| Ctrl-x 4 Ctrl-f | Creating Windows   |
| Ctrl-x 4 f      | Creating Windows   |
| Ctrl-x 4 h      | Creating Windows   |
| Ctrl-x 4 i      | Creating Windows   |
| Ctrl-x 4 `      | Creating Windows   |
| Ctrl-x 5 a      | Creating Windows   |
| Ctrl-x 5 b      | Creating Windows   |
| Ctrl-x 5 Ctrl-f | Creating Windows   |
| Ctrl-x 5 f      | Creating Windows   |
| Ctrl-x 5 h      | Creating Windows   |

---



---

|                      |                        |
|----------------------|------------------------|
| Ctrl-x 5 i           | Creating Windows       |
| Ctrl-x 5 `           | Creating Windows       |
| Ctrl-x b             | Displaying Buffers     |
| Ctrl-x Ctrl-a Ctrl-< | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-> | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-b | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-d | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-f | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-l | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-n | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-r | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-s | Debugging Programs     |
| Ctrl-x Ctrl-a Ctrl-t | Debugging Programs     |
| Ctrl-x Ctrl-b        | The Buffer Menu        |
| Ctrl-x Ctrl-f        | Commands To Load Files |
| Ctrl-x Ctrl-l        | Commands on Blocks     |
| Ctrl-x Ctrl-r        | Commands To Load Files |
| Ctrl-x Ctrl-s        | Commands To Save Files |
| Ctrl-x Ctrl-u        | Commands on Blocks     |

---

---

|               |                        |
|---------------|------------------------|
| Ctrl-x Ctrl-v | Commands To Load Files |
| Ctrl-x Ctrl-w | Commands To Save Files |
| Ctrl-x Ctrl-x | Using Marks            |
| Ctrl-x f      | Fill mode              |
| Ctrl-x h      | Marking Blocks         |
| Ctrl-x i      | Commands To Load Files |
| Ctrl-x k      | Deleting Buffers       |
| Ctrl-x o      | Other Window Commands  |
| Ctrl-x s      | Commands To Save Files |
| Ctrl-x u      | Undo                   |
| Ctrl-x `      | Finding Errors         |
| Ctrl-y        | Incremental Search     |
| Ctrl-Y        | Rectangular Blocks     |
| Ctrl-Y        | Cutting And Pasting    |
| Ctrl-y        | Cutting And Pasting    |
| Ctrl-]        | Recursive Editing      |
| Ctrl-_<br>d   | Undo                   |
| d             | Info Mode              |
| d             | The Buffer Menu        |

---

---

|            |                   |
|------------|-------------------|
| f          | Info Mode         |
| g          | Info Mode         |
| h          | Info Mode         |
| l          | Info Mode         |
| LMB-Click2 | Info Mode         |
| m          | Info Mode         |
| Meta-%     | Query Replace     |
| Meta-      | Command Arguments |
| Meta-0     | Command Arguments |
| Meta-1     | Command Arguments |
| Meta-2     | Command Arguments |
| Meta-3     | Command Arguments |
| Meta-4     | Command Arguments |
| Meta-5     | Command Arguments |
| Meta-6     | Command Arguments |
| Meta-7     | Command Arguments |
| Meta-8     | Command Arguments |
| Meta-9     | Command Arguments |
| Meta-;     | Editing Modes     |

---

---

|                |                       |
|----------------|-----------------------|
| Meta-<         | Moving Around Buffers |
| Meta->         | Moving Around Buffers |
| Meta-?         | The Buffer Prompt     |
| Meta-@         | Marking Blocks        |
| Meta-Backspace | Killing               |
| Meta-Backspace | Editing Words         |
| Meta-DEL       | Editing Words         |
| Meta-Left      | Editing Words         |
| Meta-Right     | Editing Words         |
| Meta-SPC       | Editing Characters    |
| Meta-a         | Keeping ChangeLogs    |
| Meta-b         | Editing Words         |
| Meta-c         | Editing Words         |
| Meta-d         | Killing               |
| Meta-d         | Editing Words         |
| Meta-f         | Editing Words         |
| Meta-h         | Marking Blocks        |
| Meta-i         | Moving Around Buffers |
| Meta-j         | Editing Lines         |

---

---

|                         |                       |
|-------------------------|-----------------------|
| Meta-j                  | Moving Around Buffers |
| Meta-l                  | Editing Words         |
| Meta-m                  | Moving Around Buffers |
| Meta-S                  | Text mode             |
| Meta-s                  | Text mode             |
| Meta-t                  | Editing Words         |
| Meta-TAB                | Info Mode             |
| Meta-u                  | Editing Words         |
| Meta-v                  | Moving Around Buffers |
| Meta-w                  | Killing               |
| Meta-w                  | Commands on Blocks    |
| Meta-x                  | Invoking Commands     |
| Meta-x add-autoloads    | Autoloading           |
| Meta-x recover-file     | Auto-Saving Files     |
| Meta-x remove-autoloads | Autoloading           |
| Meta-\                  | Editing Characters    |
| n                       | Info Mode             |
| n                       | Query Replace         |
| o                       | The Buffer Menu       |

---

---

|                 |                       |
|-----------------|-----------------------|
| p               | Info Mode             |
| q               | Info Mode             |
| q               | The Buffer Menu       |
| q               | Query Replace         |
| RET             | Info Mode             |
| s               | The Buffer Menu       |
| Shift-Backspace | Editing Lines         |
| Shift-DEL       | Editing Lines         |
| Shift-Down      | Moving Around Buffers |
| Shift-F1        | Using Marks           |
| Shift-F2        | Using Marks           |
| Shift-F3        | Using Marks           |
| Shift-Left      | Editing Lines         |
| Shift-Right     | Editing Lines         |
| Shift-TAB       | Moving Around Buffers |
| Shift-Up        | Moving Around Buffers |
| TAB             | Info Mode             |
| u               | Info Mode             |
| u               | The Buffer Menu       |

---

|   |                 |
|---|-----------------|
| x | The Buffer Menu |
| y | Query Replace   |
| ^ | Query Replace   |
| ~ | The Buffer Menu |

## 1.307 jade.guide/Concept Index

### Concept Index

\*\*\*\*\*

|                            |                            |
|----------------------------|----------------------------|
| Accessing compressed files | Accessing Compressed Files |
| Accessing list elements    | Accessing List Elements    |
| Alists                     | Association Lists          |
| Altering marks             | Altering Marks             |
| Anonymous functions        | Anonymous Functions        |
| Arguments, startup         | Startup Options            |
| Arithmetic Functions       | Arithmetic Functions       |
| Array functions            | Array Functions            |
| Arrays                     | Sequences                  |
| Asm mode                   | Asm mode                   |
| Association lists          | Association Lists          |
| Asynchronous processes     | Asynchronous Processes     |

---

---

|                                     |                        |
|-------------------------------------|------------------------|
| Atom                                | Cons Cells             |
| Auto-save mode                      | Auto-Save mode         |
| Auto-saves, controlling             | Controlling Auto-Saves |
| Auto-saving files                   | Auto-Saving Files      |
| Autoload forms                      | Autoload Forms         |
| Autoloading                         | Autoloading            |
| Backup files                        | Backup Files           |
| Backup files, making                | Making Backups         |
| Binding keys                        | Binding Keys           |
| Bitwise functions                   | Bitwise Functions      |
| Block marking                       | Block Marking          |
| Blocks, commands                    | Commands on Blocks     |
| Blocks, marking                     | Marking Blocks         |
| Blocks, rectangular                 | Rectangular Blocks     |
| Blocks, using                       | Using Blocks           |
| Boolean values                      | nil and t              |
| Boolean values, predicate functions | Predicate Functions    |
| Breaking out of loops               | Interrupting Jade      |
| Buffer                              | Editor Concepts        |

---



---

Buffer attributes  
Buffer Attributes

Buffer contents  
Buffer Contents

Buffer date stamps  
Buffer Date Stamps

Buffer extremes  
Buffer Extremes

Buffer glyph tables  
Buffer Glyph Tables

Buffer list  
The Buffer List

Buffer menu  
The Buffer Menu

Buffer modification counts  
Buffer Modification Counts

Buffer prompt  
The Buffer Prompt

Buffer, current  
Editor Concepts

Buffer-local variables  
Buffer-Local Variables

Buffer-local variables defined in files  
File Variables

Buffers  
Buffers

Buffers, contents of  
Buffer Attributes

Buffers, creating  
Creating Buffers

Buffers, current  
The Current Buffer

Buffers, deleting  
Deleting Buffers

Buffers, destroying  
Destroying Buffers

Buffers, displaying  
Displaying Buffers

---

---

Buffers, editing  
Editing Buffers

Buffers, editing files  
Editing Files

Buffers, file names of  
Buffer Attributes

Buffers, glyph table  
Buffer Attributes

Buffers, local variables  
Buffer Attributes

Buffers, modification counter  
Buffer Attributes

Buffers, modifications  
Modifications to Buffers

Buffers, moving around  
Moving Around Buffers

Buffers, names of  
Buffer Attributes

Buffers, other commands  
Other Buffer Commands

Buffers, positions  
Positions

Buffers, read-only  
Read-Only Buffers

Buffers, reading files  
Reading Files Into Buffers

Buffers, searching and replacing  
Searching and Replacing

Buffers, special  
Special Buffers

Buffers, tab size  
Buffer Attributes

Buffers, text  
Text

Buffers, the list of  
The Buffer List

Buffers, undo information  
Buffer Attributes

---

---

|                                |                    |
|--------------------------------|--------------------|
| Buffers, using                 | Using Buffers      |
| Buffers, writing to a file     | Writing Buffers    |
| Bugs, reporting                | Reporting Bugs     |
| Building lists                 | Building Lists     |
| C mode                         | C mode             |
| Calling commands               | Calling Commands   |
| Calling functions              | Calling Functions  |
| Catch and throw                | Catch and Throw    |
| ChangeLogs, keeping            | Keeping ChangeLogs |
| Changes since the last release | News               |
| Changes, undoing               | Undo               |
| Character images               | Character Images   |
| Character movement             | Character Movement |
| Character set, Latin-1         | Latin-1 mode       |
| Characters                     | Characters         |
| Characters, editing            | Editing Characters |
| Circular lists                 | Infinite Lists     |
| Cleanup forms                  | Cleanup Forms      |
| Client editing                 | Client Editing     |

---

---

Closing windows  
Closing Windows

Columnar blocks  
Rectangular Blocks

Command  
Editor Concepts

Command arguments  
Command Arguments

Commands  
Commands

Commands on blocks  
Commands on Blocks

Commands to load files  
Commands To Load Files

Commands to save files  
Commands To Save Files

Commands, calling  
Calling Commands

Commands, event loop  
Event Loop

Commands, example command definitions  
Example Commands

Commands, interactive declarations  
Interactive Declarations

Commands, invoking  
Invoking Commands

Commands, prefix arguments  
Prefix Arguments

Commands, window  
Other Window Commands

Comment styles  
Comment Styles

Comments  
Read Syntax

Comments, mode-specific  
Mode-Specific Comments

Comparison predicates  
Comparison Predicates

---

---

Compilation functions  
    Compilation Functions

Compilation tips  
    Compilation Tips

Compilation, disassembly of forms  
    Disassembly

Compilation, finding errors  
    Finding Errors

Compilation, running  
    Running a Compilation

Compiled Lisp  
    Compiled Lisp

Compiling macros  
    Compiling Macros

Compiling programs  
    Compiling Programs

Compressed files, accessing  
    Accessing Compressed Files

Concepts, editor  
    Editor Concepts

Conditional structures  
    Conditional Structures

Cons cells  
    Cons Cells

Constant variables  
    Constant Variables

Control Structures  
    Control Structures

Control structures, conditionals  
    Conditional Structures

Control structures, looping  
    Looping Structures

Control structures, non-local exits  
    Non-Local Exits

Control structures, sequencing  
    Sequencing Structures

Controlling auto-saves  
    Controlling Auto-Saves

---

---

|                       |                       |
|-----------------------|-----------------------|
| Controlling undo      | Controlling Undo      |
| Copying               | Copying               |
| Copying text          | Cutting And Pasting   |
| Creating buffers      | Creating Buffers      |
| Creating file objects | Creating File Objects |
| Creating glyph tables | Creating Glyph Tables |
| Creating keymaps      | Creating Keymaps      |
| Creating marks        | Creating Marks        |
| Creating symbols      | Creating Symbols      |
| Creating windows      | Creating Windows      |
| Current buffer        | The Current Buffer    |
| Current buffer        | Editor Concepts       |
| Current time          | System Time           |
| Current window        | The Current Window    |
| Current window        | Editor Concepts       |
| Cursor                | Editor Concepts       |
| Cursor position       | The Cursor Position   |
| Customisation, simple | Simple Customisation  |
| Cutting and pasting   | Cutting And Pasting   |

---

---

Data types  
Data Types

Data types, summary of  
Types Summary

Debugging  
Debugging

Debugging programs  
Debugging Programs

Default glyph table  
Buffer Glyph Tables

Defining functions  
Defining Functions

Defining macros  
Defining Macros

Defining variables  
Defining Variables

Deleting buffers  
Deleting Buffers

Deleting text  
Cutting And Pasting

Deletion functions  
Deletion Functions

Descriptions  
Descriptions

Destroying buffers  
Destroying Buffers

Destroying file objects  
Destroying File Objects

Disassembly  
Disassembly

Display, glyph tables  
Glyph Tables

Displaying buffers  
Displaying Buffers

Displaying messages  
Displaying Messages

Distribution conditions  
Copying

---

---

|                             |                       |
|-----------------------------|-----------------------|
| Editing buffers             | Editing Buffers       |
| Editing characters          | Editing Characters    |
| Editing expressions         | Editing Expressions   |
| Editing files               | Editing Files         |
| Editing lines               | Editing Lines         |
| Editing modes               | Editing Modes         |
| Editing modes, invoking     | Invoking a Mode       |
| Editing units               | Editing Units         |
| Editing words               | Editing Words         |
| Editor commands             | Commands              |
| Editor concepts             | Editor Concepts       |
| Email, my address           | Reporting Bugs        |
| Environment variables       | Environment Variables |
| Equality predicates         | Equality Predicates   |
| Errors                      | Errors                |
| Escape sequences in strings | Strings               |
| Evaluating Lisp forms       | Evaluation            |
| Evaluation                  | Evaluation            |
| Event loop                  | Event Loop            |

---



---

Event loop actions  
    Event Loop Actions

Event loop information  
    Event Loop Info

Event loop, idle actions  
    Idle Actions

Event loop, reading events  
    Reading Events

Event loop, recursive edit  
    Recursive Edits

Events, input  
    Input Events

Example commands  
    Example Commands

Example key names  
    Example Keys

Expression movement  
    Expression Movement

Expressions, editing  
    Editing Expressions

Expressions, modes-specific  
    Mode-Specific Expressions

Features  
    Features

File information  
    File Information

File names  
    File Names

File object predicates  
    File Object Predicates

File objects  
    File Objects

File objects, creating  
    Creating File Objects

File objects, destroying  
    Destroying File Objects

File objects, functions  
    Functions on File Objects

---

---

|                             |                            |
|-----------------------------|----------------------------|
| File variables              | File Variables             |
| Files                       | Files                      |
| Files, auto-saving          | Auto-Saving Files          |
| Files, backups              | Backup Files               |
| Files, closing              | Destroying File Objects    |
| Files, commands             | Other File Commands        |
| Files, editing              | Editing Files              |
| Files, loading and loading  | Loading and Saving Files   |
| Files, manipulating         | Manipulating Files         |
| Files, opening              | Creating File Objects      |
| Files, reading and writing  | Reading and Writing Files  |
| Files, reading directories  | Reading Directories        |
| Files, reading into buffers | Reading Files Into Buffers |
| Fill mode                   | Fill mode                  |
| Finding errors              | Finding Errors             |
| Font, selecting             | Other Window Commands      |
| Font, window                | Window Font                |
| Form                        | Editor Concepts            |
| Forms, autoload             | Autoload Forms             |

---

---

Forms, constant  
Self-Evaluating Forms

Forms, function call  
Function Call Forms

Forms, macro call  
Macro Call Forms

Forms, self-evaluating  
Self-Evaluating Forms

Forms, special  
Special Forms

Forms, symbol  
Symbol Forms

Forms, variable  
Symbol Forms

Function call forms  
Function Call Forms

Function exits  
Function Exits

Functions  
Functions

Functions as hooks  
Functions As Hooks

Functions on File Objects  
Functions on File Objects

Functions, anonymous  
Anonymous Functions

Functions, calling  
Calling Functions

Functions, compilation  
Compilation Functions

Functions, defining  
Defining Functions

Functions, descriptions of  
Descriptions

Functions, input  
Input Functions

Functions, lambda expressions  
Lambda Expressions

---

---

Functions, loading  
    Load Function

Functions, mapping  
    Mapping Functions

Functions, movement  
    Movement Functions

Functions, named  
    Named Functions

Functions, output  
    Output Functions

Garbage collection  
    Garbage Collection

GDB, running  
    Debugging Programs

General tips  
    General Tips

Generic mode  
    Generic mode

Global replace  
    Global Replace

Glyph positions  
    Glyph Positions

Glyph table basics  
    Glyph Table Basics

Glyph tables  
    Glyph Tables

Glyph tables, buffer  
    Buffer Glyph Tables

Glyph tables, creating  
    Creating Glyph Tables

Glyph tables, default  
    Buffer Glyph Tables

Grep, using  
    Using Grep

Gzip  
    Accessing Compressed Files

Help system  
    The Help System

---

---

Help, starting                   The Help System

Hooks                               Hooks

Hooks, functions as               Functions As Hooks

Hooks, normal                   Normal Hooks

Hooks, predefined               Standard Hooks

Iconifying windows               Iconifying Windows

Idle actions                   Idle Actions

Incremental search               Incremental Search

Indentation functions           Indentation Functions

Indentation, mode-specific       Mode-Specific Indentation

Indented-Text mode               Indented-Text mode

Infinite lists                   Infinite Lists

Info browser                   Info Mode

Info mode                       Info Mode

Initialisation procedure       Startup Procedure

Input and output               Streams

Input events                   Input Events

Input functions               Input Functions

Input streams                  Input Streams

---

---

|                          |                          |
|--------------------------|--------------------------|
| Insertion functions      | Insertion Functions      |
| Installing modes         | Installing Modes         |
| Integers                 | Numbers                  |
| Interactive declarations | Interactive Declarations |
| Interactive processes    | Interactive Processes    |
| Interactive prompts      | Prompting                |
| Interning                | Interning                |
| Interrupting Jade        | Interrupting Jade        |
| Introduction             | Introduction             |
| Introduction, Lisp       | Intro                    |
| Invocation               | Invocation               |
| Invoking a mode          | Invoking a Mode          |
| Invoking commands        | Invoking Commands        |
| Jade, using              | Using Jade               |
| Keeping ChangeLogs       | Keeping ChangeLogs       |
| Key lookup               | Key Lookup               |
| Key names                | Key Names                |
| Key names, examples      | Example Keys             |
| Key names, keys          | Keys                     |

---

---

Key names, modifiers  
    Modifiers

Keymaps  
    Keymaps

Keymaps, binding keys  
    Binding Keys

Keymaps, creating  
    Creating Keymaps

Keymaps, key lookup  
    Key Lookup

Keymaps, predefined  
    Standard Keymaps

Keymaps, prefix keys  
    Prefix Keys

Keymaps, types  
    Types of Keymap

Keys  
    Keys

Keys, event loop  
    Event Loop

Kill functions  
    Kill Functions

Killing  
    Killing

Killing buffers  
    Destroying Buffers

Killing windows  
    Killing Windows

Lambda expressions  
    Lambda Expressions

Latin-1 mode  
    Latin-1 mode

Latin1 character set  
    Character Images

Layout of programs  
    Program Layout

Licence  
    Copying

---

---

|                                   |                          |
|-----------------------------------|--------------------------|
| Line movement                     | Line Movement            |
| Lines, editing                    | Editing Lines            |
| Lisp                              | Editor Concepts          |
| Lisp forms, evaluating            | Evaluation               |
| Lisp mode                         | Lisp mode                |
| Lisp, Jade's programming language | Programming Jade         |
| List forms                        | List Forms               |
| List structure                    | List Structure           |
| Lists                             | Lists                    |
| Lists, accessing elements         | Accessing List Elements  |
| Lists, association                | Association Lists        |
| Lists, building                   | Building Lists           |
| Lists, circular                   | Infinite Lists           |
| Lists, mapping                    | Mapping Functions        |
| Lists, modifying                  | Modifying Lists          |
| Load function                     | Load Function            |
| Loading                           | Loading                  |
| Loading files                     | Loading and Saving Files |
| Loading programs                  | Loading                  |

---



---

|                                  |                     |
|----------------------------------|---------------------|
| Loading, on reference            |                     |
|                                  | Autoloading         |
| Local variables                  | Local Variables     |
| Local variables defined in files |                     |
|                                  | File Variables      |
| Looping structures               | Looping Structures  |
| Macro call forms                 | Macro Call Forms    |
| Macro expansion                  | Macro Expansion     |
| Macros                           | Macros              |
| Macros, compiling                | Compiling Macros    |
| Macros, defining                 | Defining Macros     |
| Major modes, writing             | Writing Major Modes |
| Making backups                   | Making Backups      |
| Manipulating files               | Manipulating Files  |
| Manual notation                  | Notation            |
| Mapping functions                | Mapping Functions   |
| Mark components                  | Mark Components     |
| Mark relocation                  | Mark Relocation     |
| Mark residency                   | Mark Residency      |
| Marking blocks                   | Block Marking       |
| Marking blocks                   | Marking Blocks      |

---

---

|                           |                           |
|---------------------------|---------------------------|
| Marks                     | Marks                     |
| Marks, altering           | Altering Marks            |
| Marks, creating           | Creating Marks            |
| Marks, moving to          | Moving to Marks           |
| Marks, using              | Using Marks               |
| Messages, displaying      | Displaying Messages       |
| Minor modes               | Minor Modes               |
| Minor modes, auto-save    | Auto-Save mode            |
| Minor modes, fill         | Fill mode                 |
| Minor modes, Latin-1      | Latin-1 mode              |
| Minor modes, overwrite    | Overwrite mode            |
| Minor modes, writing      | Writing Minor Modes       |
| Misc text functions       | Misc Text Functions       |
| Miscellaneous functions   | Miscellaneous Functions   |
| Mode-specific comments    | Mode-Specific Comments    |
| Mode-specific expressions | Mode-Specific Expressions |
| Mode-specific indentation | Mode-Specific Indentation |
| Modes, editing            | Editing Modes             |
| Modes, installing         | Installing Modes          |

---

---

|                                  |                          |
|----------------------------------|--------------------------|
| Modes, invoking                  | Invoking a Mode          |
| Modes, minor                     | Minor Modes              |
| Modifications to buffers         | Modifications to Buffers |
| Modifiers                        | Modifiers                |
| Modifying lists                  | Modifying Lists          |
| Movement functions               | Movement Functions       |
| Movement, character              | Character Movement       |
| Movement, expression             | Expression Movement      |
| Movement, line                   | Line Movement            |
| Movement, tab                    | Tab Movement             |
| Movement, word                   | Word Movement            |
| Moving around buffers            | Moving Around Buffers    |
| Moving to marks                  | Moving to Marks          |
| Named functions                  | Named Functions          |
| Names of files                   | File Names               |
| News                             | News                     |
| nil and t                        | nil and t                |
| Non-local exits                  | Non-Local Exits          |
| Non-local exits, catch and throw | Catch and Throw          |

---

---

Non-local exits, cleanup forms  
Cleanup Forms

Non-local exits, errors  
Errors

Non-local exits, function exits  
Function Exits

Normal hooks  
Normal Hooks

Notation  
Notation

Numbers  
Numbers

Numbers, arithmetic functions  
Arithmetic Functions

Numbers, bitwise functions  
Bitwise Functions

Numbers, predicates on  
Numeric Predicates

Numeric predicates  
Numeric Predicates

Obarrays  
Obarrays

Offsets and positions  
Positions and Offsets

Opening windows  
Opening Windows

Options, startup  
Startup Options

Other buffer commands  
Other Buffer Commands

Other file commands  
Other File Commands

Other window commands  
Other Window Commands

Output functions  
Output Functions

Output streams  
Output Streams

---

---

|                            |                        |
|----------------------------|------------------------|
| Overwrite mode             | Overwrite mode         |
| Pasting text               | Cutting And Pasting    |
| Position components        | Position Components    |
| Positions                  | Positions              |
| Positions and offsets      | Positions and Offsets  |
| Positions, buffer extremes | Buffer Extremes        |
| Positions, cursor          | The Cursor Position    |
| Positions, glyph           | Glyph Positions        |
| Positions, marks           | Marks                  |
| Predicate functions        | Predicate Functions    |
| Predicates on numbers      | Numeric Predicates     |
| Predicates, comparison     | Comparison Predicates  |
| Predicates, equality       | Equality Predicates    |
| Predicates, type           | Type Predicates        |
| Prefix arguments           | Prefix Arguments       |
| Prefix arguments, using    | Command Arguments      |
| Prefix keys                | Prefix Keys            |
| Printed representation     | Printed Representation |
| Process I/O                | Process I-O            |

---

---

|                         |                        |
|-------------------------|------------------------|
| Process information     | Process Information    |
| Process objects         | Process Objects        |
| Process states          | Process States         |
| Processes               | Processes              |
| Processes, asynchronous | Asynchronous Processes |
| Processes, interactive  | Interactive Processes  |
| Processes, signalling   | Signalling Processes   |
| Processes, synchronous  | Synchronous Processes  |
| Program layout          | Program Layout         |
| Programming Jade        | Programming Jade       |
| Programs, debugging     | Debugging Programs     |
| Programs, running       | Running a Compilation  |
| Prompt, buffer          | The Buffer Prompt      |
| Prompt, simple          | The Simple Prompt      |
| Prompt, using           | Using the Prompt       |
| Prompting               | Prompting              |
| Property lists          | Property Lists         |
| Protrams, loading       | Loading                |
| Query replace           | Query Replace          |

---

---

|                                |                            |
|--------------------------------|----------------------------|
| Quoting                        | Quoting                    |
| Read syntax                    | Read Syntax                |
| Read-only buffers              | Read-Only Buffers          |
| Reader, the Lisp               | The Lisp Reader            |
| Reading and writing files      | Reading and Writing Files  |
| Reading directories            | Reading Directories        |
| Reading events                 | Reading Events             |
| Reading files into buffers     | Reading Files Into Buffers |
| Rectangular blocks             | Rectangular Blocks         |
| Rectangular editing            | Rectangular Editing        |
| Recursive editing              | Recursive Editing          |
| Recursive edits                | Recursive Edits            |
| Regex functions                | Regex Functions            |
| Regexps                        | Regular Expressions        |
| Regular expression, definition | Editor Concepts            |
| Regular expressions            | Regular Expressions        |
| Rendering                      | Rendering                  |
| Rendering, glyph tables        | Glyph Tables               |
| Replace, global                | Global Replace             |

---

---

Replace, query  
Query Replace

Replace, search and  
Searching and Replacing

Replacing strings  
Replacing Strings

Reporting bugs  
Reporting Bugs

Requirements  
Systems Supported

Resident marks  
Mark Residency

Revision information  
Revision Information

Running a compilation  
Running a Compilation

Saving files  
Loading and Saving Files

Scope and extent  
Scope and Extent

Search and match functions, regexp functions  
Regexp Functions

Search and match functions, replacing strings  
Replacing Strings

Search and match functions, searching buffers  
Searching Buffers

Search and match functions, string matching  
String Matching

Search, incremental  
Incremental Search

Searching and matching functions  
Search and Match Functions

Searching and replacing  
Searching and Replacing

Searching buffers  
Searching Buffers

Self-evaluating forms  
Self-Evaluating Forms

---



---

|                         |                       |
|-------------------------|-----------------------|
| Sequence functions      | Sequence Functions    |
| Sequences               | Sequences             |
| Sequences, cons cells   | Cons Cells            |
| Sequencing structures   | Sequencing Structures |
| Server, Jade as a       | Client Editing        |
| Setting variables       | Setting Variables     |
| Shell                   | Shell                 |
| Shell mode              | Shell                 |
| Shell mode, programming | Interactive Processes |
| Signalling processes    | Signalling Processes  |
| Simple customisation    | Simple Customisation  |
| Simple prompt           | The Simple Prompt     |
| Special buffers         | Special Buffers       |
| Special forms           | Special Forms         |
| Standard hooks          | Standard Hooks        |
| Standard keymaps        | Standard Keymaps      |
| Starting jade           | Starting Jade         |
| Startup options         | Startup Options       |
| Startup procedure       | Startup Procedure     |

---

---

|                           |                       |
|---------------------------|-----------------------|
| Streams                   | Streams               |
| Streams, input            | Input Streams         |
| Streams, output           | Output Streams        |
| String matching           | String Matching       |
| Strings, escape sequences | Strings               |
| Style, comments           | Comment Styles        |
| Symbol attributes         | Symbol Attributes     |
| Symbol forms              | Symbol Forms          |
| Symbol syntax             | Symbol Syntax         |
| Symbols                   | Symbols               |
| Symbols, creating         | Creating Symbols      |
| Symbols, interning        | Interning             |
| Symbols, obarrays         | Obarrays              |
| Symbols, property lists   | Property Lists        |
| Synchronous processes     | Synchronous Processes |
| Syntax of objects         | Read Syntax           |
| System information        | System Information    |
| System time               | System Time           |
| t                         | nil and t             |

---

---

|                                        |                            |
|----------------------------------------|----------------------------|
| Tab movement                           | Tab Movement               |
| Texinfo mode                           | Texinfo mode               |
| Text                                   | Text                       |
| Text mode                              | Text mode                  |
| Text, buffer contents                  | Buffer Contents            |
| Text, controlling undo                 | Controlling Undo           |
| Text, deletion functions               | Deletion Functions         |
| Text, indentation functions            | Indentation Functions      |
| Text, insertion functions              | Insertion Functions        |
| Text, kill functions                   | Kill Functions             |
| Text, killing                          | Killing                    |
| Text, misc functions                   | Misc Text Functions        |
| Text, rectangular editing              | Rectangular Editing        |
| Text, regexp functions                 | Regexp Functions           |
| Text, replacing strings                | Replacing Strings          |
| Text, searching and matching functions | Search and Match Functions |
| Text, searching buffers                | Searching Buffers          |
| Text, string matching                  | String Matching            |
| Text, translation functions            | Translation Functions      |

---

---

Text, transpose functions  
    Transpose Functions

The Lisp reader  
    The Lisp Reader

Time, system  
    System Time

Tips  
    Tips

Tips, comment styles  
    Comment Styles

Tips, compilation  
    Compilation Tips

Tips, general  
    General Tips

Tips, program layout  
    Program Layout

Translation functions  
    Translation Functions

Transpose functions  
    Transpose Functions

Type predicates  
    Type Predicates

Types of keymap  
    Types of Keymap

Types summary  
    Types Summary

Undo  
    Undo

Undo, controlling  
    Controlling Undo

User information  
    User Information

Using blocks  
    Using Blocks

Using buffers  
    Using Buffers

Using grep  
    Using Grep

---

---

|                                |                        |
|--------------------------------|------------------------|
| Using Jade                     | Using Jade             |
| Using marks                    | Using Marks            |
| Using the prompt               | Using the Prompt       |
| Using windows                  | Using Windows          |
| Variable                       | Editor Concepts        |
| Variables                      | Variables              |
| Variables, buffer-local        | Buffer-Local Variables |
| Variables, constant            | Constant Variables     |
| Variables, defining            | Defining Variables     |
| Variables, descriptions of     | Descriptions           |
| Variables, local               | Local Variables        |
| Variables, scope and extent of | Scope and Extent       |
| Variables, setting             | Setting Variables      |
| Variables, void                | Void Variables         |
| Version numbers                | Revision Information   |
| Void variables                 | Void Variables         |
| Window                         | Editor Concepts        |
| Window font                    | Window Font            |
| Window information             | Window Information     |

---

---

Window, current  
Editor Concepts

Windows  
Windows

Windows, closing  
Closing Windows

Windows, creating  
Creating Windows

Windows, current  
The Current Window

Windows, cursor position  
The Cursor Position

Windows, displaying messages  
Displaying Messages

Windows, iconifying  
Iconifying Windows

Windows, killing  
Killing Windows

Windows, marking blocks  
Block Marking

Windows, opening  
Opening Windows

Windows, other commands  
Other Window Commands

Windows, rendering  
Rendering

Windows, using  
Using Windows

Word movement  
Word Movement

Words, editing  
Editing Words

Writing buffers  
Writing Buffers

Writing major modes  
Writing Major Modes

Writing minor modes  
Writing Minor Modes

---

Writing modes  
Writing Modes

Writing modes, comments  
Mode-Specific Comments

Writing modes, expressions  
Mode-Specific Expressions

Writing modes, indentation  
Mode-Specific Indentation

---