SLIME User Manual

version 3.0-alpha

Compiled: `$Date$`

Written by Luke Gorrie.
Additional contributions: Jeff Cunningham,

Table of Contents

1 Introduction

SLIME is the "Superior Lisp Interaction Mode for Emacs."

SLIME extends Emacs with support for interactive programming in Common Lisp. The features are centered around `slime-mode`, an Emacs minor-mode that complements the standard `lisp-mode`. While `lisp-mode` supports editing Lisp source files, `slime-mode` adds support for interacting with a running Common Lisp process for compilation, debugging, documentation lookup, and so on.

The `slime-mode` programming environment follows the example of Emacs's native Emacs Lisp environment. We have also included good ideas from similar systems (such as ILISP) and some new ideas of our own.

SLIME is constructed from two parts: a user-interface written in Emacs Lisp, and a supporting server program written in Common Lisp. The two sides are connected together with a socket and communicate using an RPC-like protocol.

The Lisp server is primarily written in portable Common Lisp. The required implementation-specific functionality is specified by a well-defined interface and implemented separately for each Lisp implementation. This makes SLIME readily portable.

2 Getting started

This chapter tells you how to get SLIME up and running.

2.1 Supported Platforms

SLIME supports a wide range of operating systems and Lisp implementations. SLIME runs on Unix systems, Mac OSX, and Microsoft Windows. GNU Emacs versions 21, 22, and 22 and XEmacs version 21 are supported.

The supported Lisp implementations, roughly ordered from the best-supported, are:

- CMU Common Lisp (CMUCL), 19d or newer
- Steel Bank Common Lisp (SBCL), 1.0 or newer
- Clozure Common Lisp (CCL), version 1.3 or newer
- LispWorks, version 4.3 or newer
- Allegro Common Lisp (ACL), version 6 or newer
- CLISP, version 2.35 or newer
- Armed Bear Common Lisp (ABCL)
- Corman Common Lisp, version 2.51 or newer with the patches from http://www.grumblesmurf.org/lisp/corman-patches)
- Scieneer Common Lisp (SCL), version 1.2.7 or newer
- Embedded Common Lisp (ECL)

Most features work uniformly across implementations, but some are prone to variation. These include the precision of placing compiler-note annotations, XREF support, and fancy debugger commands (like "restart frame").

2.2 Downloading SLIME

You can choose between using a released version of SLIME or accessing our CVS repository directly. You can download the latest released version from our website:

http://www.common-lisp.net/project/slime/

We recommend that users who participate in the `slime-devel` mailing list use the CVS version of the code.

2.2.1 Downloading from CVS

SLIME is available from the CVS repository on '`common-lisp.net`'. You have the option to use either the very latest code or the tagged `FAIRLY-STABLE` snapshot.

The latest version tends to have more features and fewer bugs than the `FAIRLY-STABLE` version, but it can be unstable during times of major surgery. As a rule-of-thumb recommendation we suggest that if you follow the `slime-devel` mailing list then you're better off with the latest version (we'll send a note when it's undergoing major hacking). If you don't follow the mailing list you won't know the status of the latest code, so tracking `FAIRLY-STABLE` or using a released version is the safe option.

If you checkout from CVS then remember to `cvs update` occasionally. Improvements are continually being committed, and the `FAIRLY-STABLE` tag is moved forward from time to time.

2.2.2 CVS incantations

To download SLIME you first configure your `CVSROOT` and login to the repository.

```
export CVSROOT=:pserver:anonymous@common-lisp.net:/project/slime/cvsroot
cvs login
```

(The password is `anonymous`*)*

The latest version can then be checked out with:

```
cvs checkout slime
```

Or the `FAIRLY-STABLE` version can be checked out with:

```
cvs checkout -rFAIRLY-STABLE slime
```

If you want to find out what's new since the version you're currently running, you can diff the local '`ChangeLog`' against the repository version:

```
cvs diff -rHEAD ChangeLog      # or: -rFAIRLY-STABLE
```

2.3 Installation

With a Lisp implementation that can be started from the command-line, installation just requires a few lines in your '`.emacs`':

```
(setq inferior-lisp-program "/opt/sbcl/bin/sbcl") ; your Lisp system
(add-to-list 'load-path "~/hacking/lisp/slime/")  ; your SLIME directory
(require 'slime)
(slime-setup)
```

The snippet above also appears in the '`README`' file. You can copy&paste it from there, but remember to fill in the appropriate paths.

This is the minimal configuration with the fewest frills. If the basic setup is working, you can try additional modules (Section 8.1 [Loading Contribs], page 28).

We recommend not loading the ILISP package into Emacs if you intend to use SLIME. Doing so will add a lot of extra bindings to the keymap for Lisp source files that may be confusing and may not work correctly for a Lisp process started by SLIME.

2.4 Running SLIME

SLIME is started with the Emacs command *`M-x slime`*. This uses the `inferior-lisp` package to start a Lisp process, loads and starts the Lisp-side server (known as "Swank"), and establishes a socket connection between Emacs and Lisp. Finally a REPL buffer is created where you can enter Lisp expressions for evaluation.

At this point SLIME is up and running and you can start exploring.

2.5 Setup Tuning

This section explains ways to reduce SLIME's startup time and how to configure SLIME for multiple Lisp systems.

Please proceed with this section only if your basic setup works. If you are happy with the basic setup, skip this section.

For contrib modules see Section 8.1 [Loading Contribs], page 28.

2.5.1 Autoloading

The basic setup loads SLIME always, even if you don't use SLIME. Emacs will start up a little faster if we load SLIME only on demand. To achieve that, you have to change your '`~/.emacs`' slightly:

```
(setq inferior-lisp-program "the path to your Lisp system")
(add-to-list 'load-path "the path of your 'slime' directory")
(require 'slime-autoloads)
(slime-setup)
```

The only difference compared to the basic setup is the line `(require 'slime-autoloads)`. It tells Emacs that the rest of SLIME should be loaded automatically when one of the commands *M-x slime* or *M-x slime-connect* is executed the first time.

2.5.2 Multiple Lisps

By default, the command *M-x slime* starts the program specified with `inferior-lisp-program`. If you invoke *M-x slime* with a prefix argument, Emacs prompts for the program which should be started instead. If you need that frequently or if the command involves long filenames it's more convenient to set the `slime-lisp-implementations` variable in your '`.emacs`'. For example here we define two programs:

```
(setq slime-lisp-implementations
      '((cmucl ("cmucl" "-quiet"))
        (sbcl ("/opt/sbcl/bin/sbcl") :coding-system utf-8-unix)))
```

This variable holds a list of programs and if you invoke SLIME with a negative prefix argument, *M-- M-x slime*, you can select a program from that list. The elements of the list should look like

```
(NAME (PROGRAM PROGRAM-ARGS...) &key CODING-SYSTEM INIT INIT-FUNCTION ENV)
```

`NAME` is a symbol and is used to identify the program.

`PROGRAM` is the filename of the program. Note that the filename can contain spaces.

`PROGRAM-ARGS`
: is a list of command line arguments.

`CODING-SYSTEM`
: the coding system for the connection. (see [slime-net-coding-system], page 21)

`INIT` should be a function which takes two arguments: a filename and a character encoding. The function should return a Lisp expression as a string which instructs Lisp to start the Swank server and to write the port number to the file. At startup, SLIME starts the Lisp process and sends the result of this function to Lisp's standard input. As default, `slime-init-command` is used. An example is shown in [Loading Swank faster], page 5.

`INIT-FUNCTION`
: should be a function which takes no arguments. It is called after the connection is established. (See also [slime-connected-hook], page 21.)

`ENV` specifies a list of environment variables for the subprocess. E.g.

```
(sbcl-cvs ("/home/me/sbcl-cvs/src/runtime/sbcl"
```

```
                        "--core" "/home/me/sbcl-cvs/output/sbcl.core")
                       :env ("SBCL_HOME=/home/me/sbcl-cvs/contrib/"))
```

initializes `SBCL_HOME` in the subprocess.

2.5.3 Loading Swank faster

For SBCL, we recommend that you create a custom core file with socket support and POSIX bindings included because those modules take the most time to load. To create such a core, execute the following steps:

```
shell$ sbcl
* (mapc 'require '(sb-bsd-sockets sb-posix sb-introspect sb-cltl2 asdf))
* (save-lisp-and-die "sbcl.core-for-slime")
```

After that, add something like this to your '`.emacs`':

```
(setq slime-lisp-implementations
      '((sbcl ("sbcl" "--core" "sbcl.core-for-slime"))))
```

For maximum startup speed you can include the Swank server directly in a core file. The disadvantage of this approach is that the setup is a bit more involved and that you need to create a new core file when you want to update SLIME or SBCL. The steps to execute are:

```
shell$ sbcl
* (load ".../slime/swank-loader.lisp")
* (swank-loader:dump-image "sbcl.core-with-swank")
```

Then add this to your '`.emacs`':

```
(setq slime-lisp-implementations
      '((sbcl ("sbcl" "--core" "sbcl.core-with-swank")
              :init (lambda (port-file _)
                      (format "(swank:start-server %S)\n" port-file)))))
```

Similar setups should also work for other Lisp implementations.

3 Using Slime mode

SLIME's commands are provided via `slime-mode`, a minor-mode used in conjunction with Emacs's `lisp-mode`. This chapter describes the `slime-mode` and its relatives.

3.1 User-interface conventions

To use SLIME comfortably it is important to understand a few "global" user-interface characteristics. The most important principles are described in this section.

3.1.1 Temporary buffers

Some SLIME commands create temporary buffers to display their results. Although these buffers usually have their own special-purpose major-modes, certain conventions are observed throughout.

Temporary buffers can be dismissed by pressing `q`. This kills the buffer and restores the window configuration as it was before the buffer was displayed. Temporary buffers can also be killed with the usual commands like `kill-buffer`, in which case the previous window configuration won't be restored.

Pressing *`RET`* is supposed to "do the most obvious useful thing." For instance, in an apropos buffer this prints a full description of the symbol at point, and in an XREF buffer it displays the source code for the reference at point. This convention is inherited from Emacs's own buffers for apropos listings, compilation results, etc.

Temporary buffers containing Lisp symbols use `slime-mode` in addition to any special mode of their own. This makes the usual SLIME commands available for describing symbols, looking up function definitions, and so on.

3.1.2 `*inferior-lisp*` buffer

SLIME internally uses the `comint` package to start Lisp processes. This has a few user-visible consequences, some good and some not-so-terribly. To avoid confusion it is useful to understand the interactions.

The buffer `*inferior-lisp*` contains the Lisp process's own top-level. This direct access to Lisp is useful for troubleshooting, and some degree of SLIME integration is available using the `inferior-slime-mode`. However, in normal use we recommend using the fully-integrated SLIME REPL and ignoring the `*inferior-lisp*` buffer.

3.1.3 Multithreading

If the Lisp system supports multithreading, SLIME spawns a new thread for each request, e.g., *`C-x C-e`* creates a new thread to evaluate the expression. An exception to this rule are requests from the REPL: all commands entered in the REPL buffer are evaluated in a dedicated REPL thread.

Some complications arise with multithreading and special variables. Non-global special bindings are thread-local, e.g., changing the value of a let bound special variable in one thread has no effect on the binding of the variables with the same name in other threads. This makes it sometimes difficult to change the printer or reader behaviour for new threads. The variable `swank:*default-worker-thread-bindings*` was introduced for such situtuations: instead of modifying the global value of a variable, add a binding the

`swank:*default-worker-thread-bindings*`. E.g., with the following code, new threads will read floating point values as doubles by default:

```
(push '(*read-default-float-format* . double-float)
      swank:*default-worker-thread-bindings*).
```

3.1.4 Key bindings

In general we try to make our key bindings fit with the overall Emacs style. We also have the following somewhat unusual convention of our own: when entering a three-key sequence, the final key can be pressed either with control or unmodified. For example, the `slime-describe-symbol` command is bound to *C-c C-d d*, but it also works to type *C-c C-d C-d*. We're simply binding both key sequences because some people like to hold control for all three keys and others don't, and with the two-key prefix we're not afraid of running out of keys.

There is one exception to this rule, just to trip you up. We never bind *C-h* anywhere in a key sequence, so *C-c C-d C-h* doesn't do the same thing as *C-c C-d h*. This is because Emacs has a built-in default so that typing a prefix followed by *C-h* will display all bindings starting with that prefix, so *C-c C-d C-h* will actually list the bindings for all documentation commands. This feature is just a bit too useful to clobber!

> *"Are you deliberately spiting Emacs's brilliant online help facilities? The gods will be angry!"*

This is a brilliant piece of advice. The Emacs online help facilities are your most immediate, up-to-date and complete resource for keybinding information. They are your friends:

C-h k <*key*>
: `describe-key` *"What does this key do?"*
Describes current function bound to <*key*> for focus buffer.

C-h b
: `describe-bindings` *"Exactly what bindings are available?"*
Lists the current key-bindings for the focus buffer.

C-h m
: `describe-mode` *"Tell me all about this mode"*
Shows all the available major mode keys, then the minor mode keys, for the modes of the focus buffer.

C-h l
: `view-lossage` *"Woah, what key chord did I just do?"*
Shows you the literal sequence of keys you've pressed in order.

Note: In this documentation the designation *C-h* is a *cannonical key* which might actually mean Ctrl-h, or F1, or whatever you have `help-command` bound to in your `.emacs`. Here is a common situation:

```
(global-set-key [f1]   'help-command)
(global-set-key "\C-h" 'delete-backward-char)
```

In this situation everywhere you see *C-h* in the documentation you would substitute *F1*.

You can assign or change default key bindings globally using the `global-set-key` function in your '`~/.emacs`' file like this:

```
(global-set-key "\C-c s" 'slime-selector)
```

which binds *C-c s* to the function `slime-selector`.

Alternatively, if you want to assign or change a key binding in just a particular slime mode, you can use the `global-set-key` function in your '`~/.emacs`' file like this:

```
(define-key slime-repl-mode-map (kbd "C-c ;")
            'slime-insert-balanced-comments)
```

which binds *C-c ;* to the function `slime-insert-balanced-comments` in the REPL buffer.

3.2 Evaluation commands

These commands each evaluate a Common Lisp expression in a different way. Usually they mimic commands for evaluating Emacs Lisp code. By default they show their results in the echo area, but a prefix argument causes the results to be inserted in the current buffer.

C-x C-e
M-x slime-eval-last-expression
: Evaluate the expression before point and show the result in the echo area.

C-M-x
M-x slime-eval-defun
: Evaluate the current toplevel form and show the result in the echo area. 'C-M-x' treats 'defvar' expressions specially. Normally, evaluating a 'defvar' expression does nothing if the variable it defines already has a value. But 'C-M-x' unconditionally resets the variable to the initial value specified in the 'defvar' expression. This special feature is convenient for debugging Lisp programs.

If *C-M-x* or *C-x C-e* is given a numeric argument, it inserts the value into the current buffer, rather than displaying it in the echo area.

C-c :
M-x slime-interactive-eval
: Evaluate an expression read from the minibuffer.

C-c C-r
M-x slime-eval-region
: Evaluate the region.

C-c C-p
M-x slime-pprint-eval-last-expression
: Evaluate the expression before point and pretty-print the result in a fresh buffer.

C-c E
M-x slime-edit-value
: Edit the value of a setf-able form in a new buffer '`*Edit <form>*`'. The value is inserted into a temporary buffer for editing and then set in Lisp when committed with *C-c C-c*.

C-x M-e
M-x slime-eval-last-expression-display-output
: Display the output buffer and evaluate the expression preceding point. This is useful if the expression writes something to the output stream.

C-c C-u
M-x slime-undefine-function
: Undefine the function, with `fmakunbound`, for the symbol at point.

3.3 Compilation commands

SLIME has fancy commands for compiling functions, files, and packages. The fancy part is that notes and warnings offered by the Lisp compiler are intercepted and annotated directly onto the corresponding expressions in the Lisp source buffer. (Give it a try to see what this means.)

C-c C-c
M-x slime-compile-defun
: Compile the top-level form at point. The region blinks shortly to give some feedback which part was choosen.

  With (positive) prefix argument the form is compiled with maximal debug settings. With negative prefix argument it is compiled for speed.

  The code for the region is executed after compilation. In principle, the command writes the region to a file, compiles that file, and loads the resulting code.

C-c C-k
M-x slime-compile-and-load-file
: Compile and load the current buffer's source file. If the compilation step failes, the file is not loaded. It's not always easy to tell whether the compilation failed: occasionaly you may end up in the debugger during the load step.

C-c M-k
M-x slime-compile-file
: Compile (but don't load) the current buffer's source file.

C-c C-l
M-x slime-load-file
: Load a Lisp file. This command uses the Common Lisp LOAD function.

M-x slime-compile-region
: Compile the selected region.

The annotations are indicated as underlining on source forms. The compiler message associated with an annotation can be read either by placing the mouse over the text or with the selection commands below.

M-n
M-x slime-next-note
: Move the point to the next compiler note and displays the note.

M-p
M-x slime-previous-note
: Move the point to the previous compiler note and displays the note.

C-c M-c
M-x slime-remove-notes
: Remove all annotations from the buffer.

C-x `
M-x next-error
: Visit the next-error message. This is not actually a SLIME command but SLIME creates a hidden buffer so that most of the Compilation mode commands (See

Info file ‘emacs’, node ‘Compilation Mode’) work similarily for Lisp as for batch compilers.

3.4 Completion commands

Completion commands are used to complete a symbol or form based on what is already present at point. Classical completion assumes an exact prefix and gives choices only where branches may occur. Fuzzy completion tries harder.

M-TAB
M-x slime-complete-symbol
: Complete the symbol at point. Note that three styles of completion are available in SLIME; the default is similar to normal Emacs completion (see [slime-complete-symbol-function], page 21).

3.5 Finding definitions (“Meta-Point” commands).

The familiar *M-.* command is provided. For generic functions this command finds all methods, and with some systems it does other fancy things (like tracing structure accessors to their DEFSTRUCT definition).

M-.
M-x slime-edit-definition
: Go to the definition of the symbol at point.

M-,
*M-**
M-x slime-pop-find-definition-stack
: Go back to the point where *M-.* was invoked. This gives multi-level backtracking when *M-.* has been used several times.

C-x 4 .
M-x slime-edit-definition-other-window
: Like slime-edit-definition but switchs to the other window to edit the definition in.

C-x 5 .
M-x slime-edit-definition-other-frame
: Like slime-edit-definition but opens another frame to edit the definition in.

M-x slime-edit-definition-with-etags
: Use an ETAGS table to find definition at point.

3.6 Documentation commands

SLIME’s online documentation commands follow the example of Emacs Lisp. The commands all share the common prefix *C-c C-d* and allow the final key to be modified or unmodified (see Section 3.1.4 [Key bindings], page 7.)

SPC
M-x slime-space
: The space key inserts a space, but also looks up and displays the argument list for the function at point, if there is one.

C-c C-d d
M-x slime-describe-symbol
: Describe the symbol at point.

C-c C-f
M-x slime-describe-function
: Describe the function at point.

C-c C-d a
M-x slime-apropos
: Perform an apropos search on Lisp symbol names for a regular expression match and display their documentation strings. By default the external symbols of all packages are searched. With a prefix argument you can choose a specific package and whether to include unexported symbols.

C-c C-d z
M-x slime-apropos-all
: Like `slime-apropos` but also includes internal symbols by default.

C-c C-d p
M-x slime-apropos-package
: Show apropos results of all symbols in a package. This command is for browsing a package at a high-level. With package-name completion it also serves as a rudimentary Smalltalk-ish image-browser.

C-c C-d h
M-x slime-hyperspec-lookup
: Lookup the symbol at point in the *Common Lisp Hyperspec*. This uses the familiar '`hyperspec.el`' to show the appropriate section in a web browser. The Hyperspec is found either on the Web or in `common-lisp-hyperspec-root`, and the browser is selected by `browse-url-browser-function`.

  Note: this is one case where *C-c C-d h* is *not* the same as *C-c C-d C-h*.

C-c C-d ~
M-x common-lisp-hyperspec-format
: Lookup a *format character* in the *Common Lisp Hyperspec*.

3.7 Cross-reference commands

SLIME's cross-reference commands are based on the support provided by the Lisp system, which varies widely between Lisps. For systems with no built-in XREF support SLIME queries a portable XREF package, which is taken from the *CMU AI Repository* and bundled with SLIME.

Each command operates on the symbol at point, or prompts if there is none. With a prefix argument they always prompt. You can either enter the key bindings as shown here or with the control modified on the last key, See Section 3.1.4 [Key bindings], page 7.

C-c C-w c
M-x slime-who-calls
: Show function callers.

C-c C-w w
M-x slime-calls-who
: Show all known callees.

C-c C-w r
M-x slime-who-references
: Show references to global variable.

C-c C-w b
M-x slime-who-binds
: Show bindings of a global variable.

C-c C-w s
M-x slime-who-sets
: Show assignments to a global variable.

C-c C-w m
M-x slime-who-macroexpands
: Show expansions of a macro.

M-x slime-who-specializes
: Show all known methods specialized on a class.

There are also "List callers/callees" commands. These operate by rummaging through function objects on the heap at a low-level to discover the call graph. They are only available with some Lisp systems, and are most useful as a fallback when precise XREF information is unavailable.

C-c <
M-x slime-list-callers
: List callers of a function.

C-c >
M-x slime-list-callees
: List callees of a function.

3.8 Macro-expansion commands

C-c C-m
M-x slime-macroexpand-1
: Macroexpand the expression at point once. If invoked with a prefix argument, use macroexpand instead of macroexpand-1.

C-c M-m
M-x slime-macroexpand-all
: Fully macroexpand the expression at point.

M-x slime-compiler-macroexpand-1
: Display the compiler-macro expansion of sexp at point.

M-x slime-compiler-macroexpand
: Repeatedy expamd compiler macros of sexp at point.

For additional minor-mode commands and discussion, see Section 5.2 [slime-macroexpansion-minor-mode], page 19.

3.9 Disassembly commands

C-c M-d
M-x slime-disassemble-symbol
: Disassemble the function definition of the symbol at point.

C-c C-t
M-x slime-toggle-trace-fdefinition
: Toggle tracing of the function at point. If invoked with a prefix argument, read additional information, like which particular method should be traced.

M-x slime-untrace-all
: Untrace all functions.

3.10 Abort/Recovery commands

C-c C-b
M-x slime-interrupt
: Interrupt Lisp (send `SIGINT`).

M-x slime-restart-inferior-lisp
: Restart the `inferior-lisp` process.

C-c ~
M-x slime-sync-package-and-default-directory
: Synchronize the current package and working directory from Emacs to Lisp.

C-c M-p
M-x slime-repl-set-package
: Set the current package of the REPL.

M-x slime-cd
: Set the current directory of the Lisp process. This also changes the current directory of the REPL buffer.

M-x slime-pwd
: Print the current directory of the Lisp process.

3.11 Inspector commands

The SLIME inspector is a Emacs-based alternative to the standard `INSPECT` function. The inspector presents objects in Emacs buffers using a combination of plain text, hyperlinks to related objects.

The inspector can easily be specialized for the objects in your own programs. For details see the the `inspect-for-emacs` generic function in '`swank-backend.lisp`'.

C-c I
M-x slime-inspect
: Inspect the value of an expression entered in the minibuffer.

The standard commands available in the inspector are:

RET
M-x slime-inspector-operate-on-point
: If point is on a value then recursivly call the inspcetor on that value. If point is on an action then call that action.

d
M-x slime-inspector-describe
: Describe the slot at point.

v
M-x slime-inspector-toggle-verbose
: Toggle between verbose and terse mode. Default is determined by 'swank:*inspector-verbose*'.

l
M-x slime-inspector-pop
: Go back to the previous object (return from *RET*).

n
M-x slime-inspector-next
: The inverse of *l*. Also bound to *SPC*.

q
M-x slime-inspector-quit
: Dismiss the inspector buffer.

M-RET
M-x slime-inspector-copy-down
: Store the value under point in the variable '*'. This can then be used to access the object in the REPL.

3.12 Profiling commands

The profiling commands are based on CMUCL's profiler. These are simple wrappers around functions which usually print something to the output buffer.

M-x slime-toggle-profile-fdefinition
: Toggle profiling of a function.

M-x slime-profile-package
: Profile all functions in a package.

M-x slime-unprofile-all
: Unprofile all functions.

M-x slime-profile-report
: Report profiler data.

M-x slime-profile-reset
: Reset profiler data.

M-x slime-profiled-functions
: Show list of currently profiled functions.

3.13 Shadowed Commands

C-c C-a, M-x slime-nop
C-c C-v, M-x slime-nop
: This key-binding is shadowed from inf-lisp.

3.14 Semantic indentation

SLIME automatically discovers how to indent the macros in your Lisp system. To do this the Lisp side scans all the macros in the system and reports to Emacs all the ones with `&body` arguments. Emacs then indents these specially, putting the first arguments four spaces in and the "body" arguments just two spaces, as usual.

This should "just work." If you are a lucky sort of person you needn't read the rest of this section.

To simplify the implementation, SLIME doesn't distinguish between macros with the same symbol-name but different packages. This makes it fit nicely with Emacs's indentation code. However, if you do have several macros with the same symbol-name then they will all be indented the same way, arbitrarily using the style from one of their arglists. You can find out which symbols are involved in collisions with:

```
(swank:print-indentation-lossage)
```

If a collision causes you irritation, don't have a nervous breakdown, just override the Elisp symbol's `common-lisp-indent-function` property to your taste. SLIME won't override your custom settings, it just tries to give you good defaults.

A more subtle issue is that imperfect caching is used for the sake of performance.[1]

In an ideal world, Lisp would automatically scan every symbol for indentation changes after each command from Emacs. However, this is too expensive to do every time. Instead Lisp usually just scans the symbols whose home package matches the one used by the Emacs buffer where the request comes from. That is sufficient to pick up the indentation of most interactively-defined macros. To catch the rest we make a full scan of every symbol each time a new Lisp package is created between commands – that takes care of things like new systems being loaded.

You can use *M-x slime-update-indentation* to force all symbols to be scanned for indentation information.

3.15 Reader conditional fontification

SLIME automatically evaluates reader-conditional expressions, like `#+linux`, in source buffers and "grays out" code that will be skipped for the current Lisp connection.

[1] *Of course* we made sure it was actually too slow before making the ugly optimization.

4 SLDB: the SLIME debugger

SLIME has a custom Emacs-based debugger called SLDB. Conditions signalled in the Lisp system invoke SLDB in Emacs by way of the Lisp `*DEBUGGER-HOOK*`.

SLDB pops up a buffer when a condition is signalled. The buffer displays a description of the condition, a list of restarts, and a backtrace. Commands are offered for invoking restarts, examining the backtrace, and poking around in stack frames.

4.1 Examining frames

Commands for examining the stack frame at point.

t
M-x sldb-toggle-details
: Toggle display of local variables and `CATCH` tags.

v
M-x sldb-show-source
: View the frame's current source expression. The expression is presented in the Lisp source file's buffer.

e
M-x sldb-eval-in-frame
: Evaluate an expression in the frame. The expression can refer to the available local variables in the frame.

d
M-x sldb-pprint-eval-in-frame
: Evaluate an expression in the frame and pretty-print the result in a temporary buffer.

D
M-x sldb-disassemble
: Disassemble the frame's function. Includes information such as the instruction pointer within the frame.

i
M-x sldb-inspect-in-frame
: Inspect the result of evaluating an expression in the frame.

4.2 Invoking restarts

a
M-x sldb-abort
: Invoke the `ABORT` restart.

q
M-x sldb-quit
: "Quit" – `THROW` to a tag that the top-level SLIME request-loop catches.

c
M-x sldb-continue
: Invoke the `CONTINUE` restart.

0 ... 9 Invoke a restart by number.

Restarts can also be invoked by pressing *RET* or *Mouse-2* on them in the buffer.

4.3 Navigating between frames

n, *M-x sldb-down*
p, *M-x sldb-up*
: Move between frames.

M-n, *M-x sldb-details-down*
M-p, *M-x sldb-details-up*
: Move between frames "with sugar": hide the details of the original frame and display the details and source code of the next. Sugared motion makes you see the details and source code for the current frame only.

4.4 Stepping

Stepping is not available in all implementations and works very differently in those in which it is available.

s
M-x sldb-step
: Step to the next expression in the frame. For CMUCL that means, set a breakpoint at all those code locations in the current code block which are reachable from the current code location.

x
M-x sldb-next
: [Step to the next form in the current function.]

o
M-x sldb-next
: [Stop single-stepping temporarily, but resume it once the current function returns.]

4.5 Miscellaneous Commands

r
M-x sldb-restart-frame
: Restart execution of the frame with the same arguments it was originally called with. (This command is not available in all implementations.)

R
M-x sldb-return-from-frame
: Return from the frame with a value entered in the minibuffer. (This command is not available in all implementations.)

B
M-x sldb-break-with-default-debugger
: Exit SLDB and debug the condition using the Lisp system's default debugger.

`:`
`M-x slime-interactive-eval`
: Evaluate an expression entered in the minibuffer.

5 Misc

5.1 slime-selector

The `slime-selector` command is for quickly switching to important buffers: the REPL, SLDB, the Lisp source you were just hacking, etc. Once invoked the command prompts for a single letter to specify which buffer it should display. Here are some of the options:

? A help buffer listing all `slime-selectors`'s available buffers.

r The REPL buffer for the current SLIME connection.

d The most recently activated SLDB buffer for the current connection.

l The most recently visited `lisp-mode` source buffer.

s The `*slime-scratch*` buffer (see [slime-scratch], page 38).

`slime-selector` doesn't have a key binding by default but we suggest that you assign it a global one. You can bind it to *C-c s* like this:

```
(global-set-key "\C-cs" 'slime-selector)
```

And then you can switch to the REPL from anywhere with *C-c s r*.

The macro `def-slime-selector-method` can be used to define new buffers for `slime-selector` to find.

5.2 slime-macroexpansion-minor-mode

Within a slime macroexpansion buffer some extra commands are provided (these commands are always available but are only bound to keys in a macroexpansion buffer).

C-c C-m
M-x slime-macroexpand-1-inplace
: Just like slime-macroexpand-1 but the original form is replaced with the expansion.

g
M-x slime-macroexpand-1-inplace
: The last macroexpansion is performed again, the current contents of the macroexpansion buffer are replaced with the new expansion.

q
M-x slime-temp-buffer-quit
: Close the expansion buffer.

5.3 Multiple connections

SLIME is able to connect to multiple Lisp processes at the same time. The *M-x slime* command, when invoked with a prefix argument, will offer to create an additional Lisp process if one is already running. This is often convenient, but it requires some understanding to make sure that your SLIME commands execute in the Lisp that you expect them to.

Some buffers are tied to specific Lisp processes. Each Lisp connection has its own REPL buffer, and all expressions entered or SLIME commands invoked in that buffer are sent

to the associated connection. Other buffers created by SLIME are similarly tied to the connections they originate from, including SLDB buffers, apropos result listings, and so on. These buffers are the result of some interaction with a Lisp process, so commands in them always go back to that same process.

Commands executed in other places, such as `slime-mode` source buffers, always use the "default" connection. Usually this is the most recently established connection, but this can be reassigned via the "connection list" buffer:

C-c C-x c
M-x slime-list-connections
: Pop up a buffer listing the established connections.

C-c C-x t
M-x slime-list-threads
: Pop up a buffer listing the current threads.

The buffer displayed by `slime-list-connections` gives a one-line summary of each connection. The summary shows the connection's serial number, the name of the Lisp implementation, and other details of the Lisp process. The current "default" connection is indicated with an asterisk.

The commands available in the connection-list buffer are:

RET
M-x slime-goto-connection
: Pop to the REPL buffer of the connection at point.

d
M-x slime-connection-list-make-default
: Make the connection at point the "default" connection. It will then be used for commands in `slime-mode` source buffers.

g
M-x slime-update-connection-list
: Update the connection list in the buffer.

q
M-x slime-temp-buffer-quit
: Quit the connection list (kill buffer, restore window configuration).

R
M-x slime-restart-connection-at-point
: Restart the Lisp process for the connection at point.

M-x slime-connect
: Connect to a running Swank server.

M-x slime-disconnect
: Disconnect all connections.

M-x slime-abort-connection
: Abort the current attempt to connect.

6 Customization

6.1 Emacs-side

The Emacs part of SLIME can be configured with the Emacs `customize` system, just use *`M-x customize-group slime RET`*. Because the customize system is self-describing, we only cover a few important or obscure configuration options here in the manual.

`slime-truncate-lines`
: The value to use for `truncate-lines` in line-by-line summary buffers popped up by SLIME. This is `t` by default, which ensures that lines do not wrap in backtraces, apropos listings, and so on. It can however cause information to spill off the screen.

`slime-complete-symbol-function`
: The function to use for completion of Lisp symbols. Three completion styles are available: `slime-simple-complete-symbol`, `slime-complete-symbol*` (see Section 8.5 [Compound Completion], page 31), and `slime-fuzzy-complete-symbol` (see Section 8.6 [Fuzzy Completion], page 32).

  The default is `slime-simple-complete-symbol`, which completes in the usual Emacs way.

`slime-filename-translations`
: This variable controls filename translation between Emacs and the Lisp system. It is useful if you run Emacs and Lisp on separate machines which don't share a common file system or if they share the filessytem but have different layouts, as is the case with SMB-based file sharing.

`slime-net-coding-system`
: If you want to transmit Unicode characters between Emacs and the Lisp system, you should customize this variable. E.g., if you use SBCL, you can set:

```
(setq slime-net-coding-system 'utf-8-unix)
```

  To actually display Unicode characters you also need appropriate fonts, otherwise the characters will be rendered as hollow boxes. If you are using Allegro CL and GNU Emacs, you can also use `emacs-mule-unix` as coding system. GNU Emacs has often nicer fonts for the latter encoding. (Different encodings can be used for different Lisps, see Section 2.5.2 [Multiple Lisps], page 4.)

6.1.1 Hooks

`slime-mode-hook`
: This hook is run each time a buffer enters `slime-mode`. It is most useful for setting buffer-local configuration in your Lisp source buffers. An example use is to enable `slime-autodoc-mode` (see Section 8.7 [slime-autodoc-mode], page 33).

`slime-connected-hook`
: This hook is run when SLIME establishes a connection to a Lisp server. An example use is to create a Typeout frame (See Section 8.13 [Typeout frames], page 37.)

`sldb-hook`
: This hook is run after SLDB is invoked. The hook functions are called from the SLDB buffer after it is initialized. An example use is to add `sldb-print-condition` to this hook, which makes all conditions debugged with SLDB be recorded in the REPL buffer.

6.2 Lisp-side (Swank)

The Lisp server side of SLIME (known as "Swank") offers several variables to configure. The initialization file '`~/.swank.lisp`' is automatically evaluated at startup and can be used to set these variables.

6.2.1 Communication style

The most important configurable is `SWANK:*COMMUNICATION-STYLE*`, which specifies the mechanism by which Lisp reads and processes protocol messages from Emacs. The choice of communication style has a global influence on SLIME's operation.

The available communication styles are:

`NIL`
: This style simply loops reading input from the communication socket and serves SLIME protocol events as they arise. The simplicity means that the Lisp cannot do any other processing while under SLIME's control.

`:FD-HANDLER`
: This style uses the classical Unix-style "`select()`-loop." Swank registers the communication socket with an event-dispatching framework (such as `SERVE-EVENT` in CMUCL and SBCL) and receives a callback when data is available. In this style requests from Emacs are only detected and processed when Lisp enters the event-loop. This style is simple and predictable.

`:SIGIO`
: This style uses *signal-driven I/O* with a `SIGIO` signal handler. Lisp receives requests from Emacs along with a signal, causing it to interrupt whatever it is doing to serve the request. This style has the advantage of responsiveness, since Emacs can perform operations in Lisp even while it is busy doing other things. It also allows Emacs to issue requests concurrently, e.g. to send one long-running request (like compilation) and then interrupt that with several short requests before it completes. The disadvantages are that it may conflict with other uses of `SIGIO` by Lisp code, and it may cause untold havoc by interrupting Lisp at an awkward moment.

`:SPAWN`
: This style uses multiprocessing support in the Lisp system to execute each request in a separate thread. This style has similar properties to `:SIGIO`, but it does not use signals and all requests issued by Emacs can be executed in parallel.

The default request handling style is chosen according to the capabilities of your Lisp system. The general order of preference is `:SPAWN`, then `:SIGIO`, then `:FD-HANDLER`, with `NIL` as a last resort. You can check the default style by calling `SWANK-BACKEND:PREFERRED-COMMUNICATION-STYLE`. You can also override the default by setting `SWANK:*COMMUNICATION-STYLE*` in your Swank init file.

6.2.2 Other configurables

These Lisp variables can be configured via your '`~/.swank.lisp`' file:

`SWANK:*CONFIGURE-EMACS-INDENTATION*`
: This variable controls whether indentation styles for `&body`-arguments in macros are discovered and sent to Emacs. It is enabled by default.

`SWANK:*GLOBALLY-REDIRECT-IO*`
: When true this causes the standard streams (`*standard-output*`, etc) to be globally redirected to the REPL in Emacs. When `NIL` (the default) these streams are only temporarily redirected to Emacs using dynamic bindings while handling requests. Note that `*standard-input*` is currently never globally redirected into Emacs, because it can interact badly with the Lisp's native REPL by having it try to read from the Emacs one.

`SWANK:*GLOBAL-DEBUGGER*`
: When true (the default) this causes `*DEBUGGER-HOOK*` to be globally set to `SWANK:SWANK-DEBUGGER-HOOK` and thus for SLIME to handle all debugging in the Lisp image. This is for debugging multithreaded and callback-driven applications.

`SWANK:*SLDB-PRINTER-BINDINGS*`
`SWANK:*MACROEXPAND-PRINTER-BINDINGS*`
`SWANK:*SWANK-PPRINT-BINDINGS*`
: These variables can be used to customize the printer in various situations. The values of the variables are association lists of printer variable names with the corresponding value. E.g., to enable the pretty printer for formatting backtraces in SLDB, you can use:

```
(push '(*print-pretty* . t) swank:*sldb-printer-bindings*).
```

`SWANK:*USE-DEDICATED-OUTPUT-STREAM*`
: This variable controls whether to use an unsafe efficiency hack for sending printed output from Lisp to Emacs. The default is `nil`, don't use it, and is strongly recommended to keep.

  When `t`, a separate socket is established solely for Lisp to send printed output to Emacs through, which is faster than sending the output in protocol-messages to Emacs. However, as nothing can be guaranteed about the timing between the dedicated output stream and the stream of protocol messages, the output of a Lisp command can arrive before or after the corresponding REPL results. Thus output and REPL results can end up in the wrong order, or even interleaved, in the REPL buffer. Using a dedicated output stream also makes it more difficult to communicate to a Lisp running on a remote host via SSH (see Section 7.1 [Connecting to a remote lisp], page 25).

`SWANK:*DEDICATED-OUTPUT-STREAM-PORT*`
: When `*USE-DEDICATED-OUTPUT-STREAM*` is `t` the stream will be opened on this port. The default value, `0`, means that the stream will be opened on some random port.

`SWANK:*LOG-EVENTS*`

Setting this variable to `t` causes all protocol messages exchanged with Emacs to be printed to `*TERMINAL-IO*`. This is useful for low-level debugging and for observing how SLIME works "on the wire." The output of `*TERMINAL-IO*` can be found in your Lisp system's own listener, usually in the buffer `*inferior-lisp*`.

7 Tips and Tricks

7.1 Connecting to a remote lisp

One of the advantages of the way SLIME is implemented is that we can easily run the Emacs side (slime.el) on one machine and the lisp backend (swank) on another. The basic idea is to start up lisp on the remote machine, load swank and wait for incoming slime connections. On the local machine we start up emacs and tell slime to connect to the remote machine. The details are a bit messier but the underlying idea is that simple.

7.1.1 Setting up the lisp image

When you want to load swank without going through the normal, Emacs based, process just load the '`swank-loader.lisp`' file. Just execute

```
(load "/path/to/swank-loader.lisp")
```

inside a running lisp image[1]. Now all we need to do is startup our swank server. The first example assumes we're using the default settings.

```
(swank:create-server)
```

Since we're going to be tunneling our connection via ssh[2] and we'll only have one port open we want to tell swank to not use an extra connection for output (this is actually the default in current SLIME):

```
(setf swank:*use-dedicated-output-stream* nil)
```

If you need to do anything particular (like be able to reconnect to swank after you're done), look into `swank:create-server`'s other arguments. Some of these arguments are

`:PORT` Port number for the server to listen on (default: 4005).

`:STYLE` See See Section 6.2.1 [Communication style], page 22.

`:DONT-CLOSE`
Boolean indicating if the server will continue to accept connections after the first one (default: `NIL`). For "long-running" lisp processes to which you want to be able to connect from time to time, specify `:dont-close t`

`:CODING-SYSTEM`
String designating the encoding to be used to communicate between the Emacs and Lisp.

So the more complete example will be

```
(swank:create-server :port 4005  :dont-close t :coding-system "utf-8-unix")
```

On the emacs side you will use something like

```
(setq slime-net-coding-system 'utf-8-unix)
(slime-connect "127.0.0.1" 4005))
```

to connect to this lisp image from the same machine.

---

[1] SLIME also provides an ASDF system definiton which does the same thing

[2] there is a way to connect without an ssh tunnel, but it has the side-effect of giving the entire world access to your lisp image, so we're not going to talk about it

7.1.2 Setting up Emacs

Now we need to create the tunnel between the local machine and the remote machine.

```
ssh -L4005:127.0.0.1:4005 username@remote.example.com
```

That ssh invocation creates an ssh tunnel between the port 4005 on our local machine and the port 4005 on the remote machine[3].

Finally we can start SLIME:

```
M-x slime-connect RET RET
```

The *RET RET* sequence just means that we want to use the default host (`127.0.0.1`) and the default port (`4005`). Even though we're connecting to a remote machine the ssh tunnel fools Emacs into thinking it's actually `127.0.0.1`.

7.1.3 Setting up pathname translations

One of the main problems with running swank remotely is that Emacs assumes the files can be found using normal filenames. if we want things like `slime-compile-and-load-file` (*C-c C-k*) and `slime-edit-definition` (*M-.*) to work correctly we need to find a way to let our local Emacs refer to remote files.

There are, mainly, two ways to do this. The first is to mount, using NFS or similar, the remote machine's hard disk on the local machine's file system in such a fashion that a filename like '`/opt/project/source.lisp`' refers to the same file on both machines. Unfortunetly NFS is usually slow, often buggy, and not always feasable, fortunetely we have an ssh connection and Emacs' `tramp-mode` can do the rest. (See See Info file '`tramp`', node '`Top`'.)

What we do is teach Emacs how to take a filename on the remote machine and translate it into something that tramp can understand and access (and vice-versa). Assuming the remote machine's host name is `remote.example.com`, `cl:machine-instance` returns "remote" and we login as the user "user" we can use SLIME's built-in mechanism to setup the proper transaltions by simply doing:

```
(push (slime-create-filename-translator :machine-instance "remote.example.com"
                                        :remote-host "remote"
                                        :username "user")
      slime-filename-translations)
```

7.2 Globally redirecting all IO to the REPL

By default SLIME does not change `*standard-output*` and friends outside of the REPL. If you have any other threads which call `format`, `write-string`, etc. that output will be seen only in the `*inferior-lisp*` buffer or on the terminal, more often than not this is inconvenient. So, if you want code such as this:

```
(run-in-new-thread
  (lambda ()
    (write-line "In some random thread.~%" *standard-output*)))
```

[3] By default swank listens for incoming connections on port 4005, had we passed a `:port` parameter to `swank:create-server` we'd be using that port number instead

to send its output to SLIME's repl buffer, as opposed to `*inferior-lisp*`, set `swank:*globally-redirect-io*` to T.

Note that the value of this variable is only checked when swank accepts the connection so you should set it via '`~/.swank.lisp`'. Otherwise you will need to call `swank::globally-redirect-io-to-connection` yourself, but you shouldn't do that unless you know what you're doing.

7.3 Connecting to SLIME automatically

To make SLIME connect to your lisp whenever you open a lisp file just add this to your '`.emacs`':

```
(add-hook 'slime-mode-hook
          (lambda ()
            (unless (slime-connected-p)
              (save-excursion (slime)))))
```

8 Contributed Packages

In version 3.0 we moved some functionality to separate packages. This chapter tells you how to load contrib modules and describes what the particular packages do.

8.1 Loading Contrib Packages

Contrib packages aren't loaded by default. You have to modify your setup a bit so that Emacs knows where to find them and which of them to load. Generally, you should call `slime-setup` with the list of package-names that you want to use. For example, a setup to load the `slime-scratch` and `slime-editing-commands` packages looks like:

```
(setq inferior-lisp-program "/opt/sbcl/bin/sbcl") ; your Lisp system
(add-to-list 'load-path "~/hacking/lisp/slime/")  ; your SLIME directory
(require 'slime-autoloads)
(slime-setup '(slime-scratch slime-editing-commands))
```

After starting SLIME, the commands of both packages should be available.

8.2 REPL: the "top level"

SLIME uses a custom Read-Eval-Print Loop (REPL, also known as a "top level", or listener). The REPL user-interface is written in Emacs Lisp, which gives more Emacs-integration than the traditional `comint`-based Lisp interaction:

- Conditions signalled in REPL expressions are debugged with SLDB.
- Return values are distinguished from printed output by separate Emacs faces (colours).
- Emacs manages the REPL prompt with markers. This ensures that Lisp output is inserted in the right place, and doesn't get mixed up with user input.

To load the REPL call `(slime-setup '(slime-repl))` in your `./emacs`.

C-c C-z
M-x slime-switch-to-output-buffer
: Select the output buffer, preferably in a different window.

C-c C-y
M-x slime-call-defun
: Insert a call to the function defined around point into the REPL.

8.2.1 REPL commands

RET
M-x slime-repl-return
: Evaluate the current input in Lisp if it is complete. If incomplete, open a new line and indent. If a prefix argument is given then the input is evaluated without checking for completeness.

C-RET
M-x slime-repl-closing-return
: Close any unmatched parenthesis and then evaluate the current input in Lisp. Also bound to *M-RET*.

C-j
M-x slime-repl-newline-and-indent
: Open and indent a new line.

C-a
M-x slime-repl-bol
: Go to the beginning of the line, but stop at the REPL prompt.

C-c C-c
M-x slime-interrupt
: Interrupt the Lisp process with `SIGINT`.

C-c M-o
M-x slime-repl-clear-buffer
: Clear the entire buffer, leaving only a prompt.

C-c C-o
M-x slime-repl-clear-output
: Remove the output and result of the previous expression from the buffer.

8.2.2 Input navigation

The input navigation (a.k.a. history) commands are modelled after `coming`-mode. Be careful if you are used to Bash-like keybindings: *M-p* and *M-n* use the current input as search pattern and only work Bash-like if the current line is empty. *C-<up>* and *C-<up>* work like the up and down keys in Bash.

C-<up>, M-x slime-repl-forward-input
C-<down>, M-x slime-repl-backward-input
: Go to the next/previous history item.

M-n, M-x slime-repl-next-input
M-p, M-x slime-repl-previous-input
: Search the next/previous item in the command history using the current input as search pattern. If *M-n*/*M-n* is typed two times in a row, the second invocation uses the same search pattern (even if the current input has changed).

M-s, M-x slime-repl-next-matching-input
M-r, M-x slime-repl-previous-matching-input
: Search forward/reverse through command history with regex

C-c C-n, M-x slime-repl-next-prompt
C-c C-p, M-x slime-repl-previous-prompt
: Move between the current and previous prompts in the REPL buffer. Pressing RET on a line with old input copies that line to the newest prompt.

The variable `slime-repl-wrap-history` controls wrap around behaviour, i.e. whether cycling should restart at the beginning of the history if the end is reached.

8.2.3 Shortcuts

"Shortcuts" are a special set of REPL commands that are invoked by name. To invoke a shortcut you first press `,` (comma) at the REPL prompt and then enter the shortcut's name when prompted.

Shortcuts deal with things like switching between directories and compiling and loading Lisp systems. The set of shortcuts is listed below, and you can also use the `help` shortcut to list them interactively.

change-directory (aka !d, cd)
: Change the current directory.

change-package (aka !p)
: Change the current package.

compile-and-load (aka cl)
: Compile (if neccessary) and load a lisp file.

defparameter (aka !)
: Define a new global, special, variable.

help (aka ?)
: Display the help.

pop-directory (aka -d)
: Pop the current directory.

pop-package (aka -p)
: Pop the top of the package stack.

push-directory (aka +d, pushd)
: Push a new directory onto the directory stack.

push-package (aka +p)
: Push a package onto the package stack.

pwd
: Show the current directory.

quit
: Quit the current Lisp.

resend-form
: Resend the last form.

restart-inferior-lisp
: Restart *inferior-lisp* and reconnect SLIME.

sayoonara
: Quit all Lisps and close all SLIME buffers.

8.3 Multiple REPLs

The `slime-mrepl` package adds support for multiple listener buffers. The command *M-x slime-open-listener* creates a new buffer. In a multi-threaded Lisp, each listener is associated with a separate thread. In a single-threaded Lisp it's also possible to create multiple listener buffers but the commands are executed sequentially by the same process.

8.4 inferior-slime-mode

The `inferior-slime-mode` is a minor mode is intended to use with the `*inferior-lisp*` lisp buffer. It provides some of the SLIME commands, like symbol completion and documentation lookup. It also tracks the current directory of the Lisp process. To install it, add something like this to user '`.emacs`':

```
(slime-setup '(inferior-slime-mode))
```

M-x inferior-slime-mode
: Turns inferior-slime-mode on or off.

The variable `inferior-slime-mode-map` contains the extra keybindings.

8.5 Compound Completion

The package `slime-c-p-c` provides a different symbol completion algorithm, which performs completion "in parallel" over the hyphen-delimited sub-words of a symbol name.[1] Formally this means that "`a-b-c`" can complete to any symbol matching the regular expression "`^a.*-b.*-c.*`" (where "dot" matches anything but a hyphen). Examples give a more intuitive feeling:

- `m-v-b` completes to `multiple-value-bind`.
- `w-open` is ambiguous: it completes to either `with-open-file` or `with-open-stream`. The symbol is expanded to the longest common completion (`with-open-`) and the point is placed at the first point of ambiguity, which in this case is the end.
- `w--stream` completes to `with-open-stream`.

The variable `slime-c-p-c-unambiguous-prefix-p` specifies where point should be placed after completion. E.g. the possible completions for `f-o` are `finish-output` and `force-output`. By the default point is moved after the `f`, because that is the unambigous prefix. If `slime-c-p-c-unambiguous-prefix-p` is nil, point moves to the end of the inserted text, after the `o` in this case.

In addition, `slime-c-p-c` provides completion for character names (mostly useful for Unicode-aware implementations):

```
CL-USER> #\Sp<TAB>
```

Here SLIME will usually complete the character to `#\Space`, but in a Unicode-aware implementation, this might provide the following completions:

```
Space                          Space
Sparkle                        Spherical_Angle
Spherical_Angle_Opening_Left   Spherical_Angle_Opening_Up
```

The package `slime-c-p-c` also provides context-sensitive completion for keywords. Example:

```
CL-USER> (find 1 '(1 2 3) :s<TAB>
```

Here SLIME will complete `:start`, rather than suggesting all ever-interned keywords starting with `:s`.

[1] This style of completion is modelled on '`completer.el`' by Chris McConnell. That package is bundled with ILISP.

C-c C-s
M-x slime-complete-form
: Looks up and inserts into the current buffer the argument list for the function at point, if there is one. More generally, the command completes an incomplete form with a template for the missing arguments. There is special code for discovering extra keywords of generic functions and for handling `make-instance`, `defmethod`, and many other functions. Examples:

```
(subseq "abc" <C-c C-s>
  --inserts--> start [end])
(find 17 <C-c C-s>
  --inserts--> sequence :from-end from-end :test test
               :test-not test-not :start start :end end
               :key key)
(find 17 '(17 18 19) :test #'= <C-c C-s>
  --inserts--> :from-end from-end
               :test-not test-not :start start :end end
               :key key)
(defclass foo () ((bar :initarg :bar)))
(defmethod print-object <C-c C-s>
  --inserts-->   (object stream)
              body...)
(defmethod initialize-instance :after ((object foo) &key blub))
(make-instance 'foo <C-c C-s>
  --inserts--> :bar bar :blub blub initargs...)
```

8.6 Fuzzy Completion

The package `slime-fuzzy` implements yet another symbol completion heuristic.

[Somebody please describe what the algorithm actually does]

It attempts to complete a symbol all at once, instead of in pieces. For example, "mvb" will find "`multiple-value-bind`" and "norm-df" will find "`least-positive-normalized-double-float`".

The algorithm tries to expand every character in various ways and rates the list of possible completions with the following heuristic.

Letters are given scores based on their position in the string. Letters at the beginning of a string or after a prefix letter at the beginning of a string are scored highest. Letters after a word separator such as #\- are scored next highest. Letters at the end of a string or before a suffix letter at the end of a string are scored medium, and letters anywhere else are scored low.

If a letter is directly after another matched letter, and its intrinsic value in that position is less than a percentage of the previous letter's value, it will use that percentage instead.

Finally, a small scaling factor is applied to favor shorter matches, all other things being equal.

C-c M-i
M-x slime-fuzzy-complete-symbol
: Presents a list of likely completions to choose from for an abbreviation at point. If you set the variable `slime-complete-symbol-function` to this command, fuzzy completion will also be used for *M-TAB*.

8.7 slime-autodoc-mode

Autodoc mode is an additional minor-mode for automatically showing information about symbols near the point. For function names the argument list is displayed, and for global variables, the value. This is a clone of `eldoc-mode` for Emacs Lisp.

The mode can be enabled by default in the `slime-setup` call of your `~/.emacs`:

```
(slime-setup '(slime-autodoc))
```

M-x slime-arglist NAME
: Show the argument list of the function NAME.

M-x slime-autodoc-mode
: Toggles autodoc-mode on or off according to the argument, and toggles the mode when invoked without argument.

If the variable `slime-use-autodoc-mode` is set (default), Emacs starts a timer, otherwise the information is only displayed after pressing SPC.

8.8 ASDF

ASDF is a popular "system construction tool". The package `slime-asdf` provides some commands to load and compile such systems from Emacs. ASDF itself is not included with SLIME; you have to load that yourself into your Lisp. In particular, you must load ASDF before you connect, otherwise you will get errors about missing symbols.

M-x slime-load-system NAME
: Compile and load an ASDF system. The default system name is taken from the first file matching *.asd in the current directory.

The package also installs some new REPL shortcuts (see Section 8.2.3 [Shortcuts], page 29):

load-system
: Compile (as needed) and load an ASDF system.

compile-system
: Compile (but not load) an ASDF system.

force-compile-system
: Recompile (but not load) an ASDF system.

force-load-system
: Recompile and load an ASDF system.

8.9 Banner

The package `slime-banner` installs a window header line (See Info file '`elisp`', node '`Header Lines`'.) in the REPL buffer. It also runs an animation at startup.

By setting the variable `slime-startup-animation` to nil you can disable the animation respectivly with the variable `slime-header-line-p` the header line.

8.10 Editing Commands

The package `slime-editing-commands` provides some commands to edit Lisp expressions.

C-c M-q
M-x slime-reindent-defun
: Re-indents the current defun, or refills the current paragraph. If point is inside a comment block, the text around point will be treated as a paragraph and will be filled with `fill-paragraph`. Otherwise, it will be treated as Lisp code, and the current defun will be reindented. If the current defun has unbalanced parens, an attempt will be made to fix it before reindenting.

M-x slime-close-all-parens-in-sexp
: Balance parentheses of open s-expressions at point. Insert enough right parentheses to balance unmatched left parentheses. Delete extra left parentheses. Reformat trailing parentheses Lisp-stylishly.

  If REGION is true, operate on the region. Otherwise operate on the top-level sexp before point.

M-x slime-insert-balanced-comments
: Insert a set of balanced comments around the s-expression containing the point. If this command is invoked repeatedly (without any other command occurring between invocations), the comment progressively moves outward over enclosing expressions. If invoked with a positive prefix argument, the s-expression arg expressions out is enclosed in a set of balanced comments.

M-C-a
M-x slime-beginning-of-defun
M-C-e
M-x slime-end-of-defun

8.11 Fancy Inspector

An alternative to default inspector is provided by the package 'slime-fancy-inspector'. This inspector knows a lot about CLOS objects and methods. It provides many "actions" that can be selected to invoke Lisp code on the inspected object. For example, to present a generic function the inspector shows the documentation in plain text and presents each method with both a hyperlink to inspect the method object and a "remove method" action that you can invoke interactively. The key-bindings are the same as for the basic inspector (see Section 3.11 [Inspector], page 13).

8.12 Presentations

A "presentation"[2] in SLIME is a region of text associated with a Lisp object. Right-clicking on the text brings up a menu with operations for the particular object. Some operations, like inspecting, are available for all objects, but the object may also have specialized operations. For instance, pathnames have a dired operation.

More importantly, it is possible to cut and paste presentations (i.e., Lisp objects, not just their printed presentation), using all standard Emacs commands. This way it is possible to cut and paste the results of previous computations in the REPL. This is of particular importance for unreadable objects.

The package `slime-presentations` installs presentations in the REPL, i.e. the results of evaluation commands become presentations. In this way, presentations generalize the use of the standard Common Lisp REPL history variables `*`, `**`, `***`. Example:

```
CL-USER> (find-class 'standard-class)
#<STANDARD-CLASS STANDARD-CLASS>
CL-USER>
```

Presentations appear in red color in the buffer. (In this manual, we indicate the presentations *like this*.) Using standard Emacs commands, the presentation can be copied to a new input in the REPL:

```
CL-USER> (eql '#<STANDARD-CLASS STANDARD-CLASS> '#<STANDARD-CLASS STANDARD-
CLASS>)
T
```

When you copy an incomplete presentation or edit the text within a presentation, the presentation changes to plain text, losing the association with a Lisp object. In the buffer, this is indicated by changing the color of the text from red to black. This can be undone.

Presentations are also available in the inspector (all inspectable parts are presentations) and the debugger (all local variables are presentations). This makes it possible to evaluate expressions in the REPL using objects that appear in local variables of some active debugger frame; this can be more convenient than using `M-x sldb-eval-in-frame`. **Warning:** The presentations that stem from the inspector and debugger are only valid as long as the corresponding buffers are open. Using them later can cause errors or confusing behavior.

For some Lisp implementations you can also install the package `slime-presentation-streams`, which enables presentations on the Lisp `*standard-output*` stream and similar streams. This means that not only results of computations, but also some objects that are printed to the standard output (as a side-effect of the computation) are associated with presentations. Currently, all unreadable objects and pathnames get printed as presentations.

```
CL-USER> (describe (find-class 'standard-object))
#<STANDARD-CLASS STANDARD-OBJECT> is an instance of
    #<STANDARD-CLASS STANDARD-CLASS>:
 The following slots have :INSTANCE allocation:
  PLIST                  NIL
  FLAGS                  1
```

[2] Presentations are a feature originating from the Lisp machines. It was possible to define `present` methods specialized to various devices, e.g. to draw an object to bitmapped screen or to write some text to a character stream.

```
DIRECT-METHODS          ((#<STANDARD-METHOD
                              SWANK::ALL-SLOTS-FOR-INSPECTOR
                              (STANDARD-OBJECT T)>
...
```

Again, this makes it possible to inspect and copy-paste these objects.

In addition to the standard Emacs commands, there are several keyboard commands, a menu-bar menu, and a context menu to operate on presentations. We describe the keyboard commands below; they are also shown in the menu-bar menu.

C-c C-v SPC
M-x slime-mark-presentation
: If point is within a presentation, move point to the beginning of the presentation and mark to the end of the presentation. This makes it possible to copy the presentation.

C-c C-v w
M-x slime-copy-presentation-at-point-to-kill-ring
: If point is within a presentation, copy the surrounding presentation to the kill ring.

C-c C-v r
M-x slime-copy-presentation-at-point-to-repl
: If point is within a presentation, copy the surrounding presentation to the REPL.

C-c C-v d
M-x slime-describe-presentation-at-point
: If point is within a presentation, describe the associated object.

C-c C-v i
M-x slime-inspect-presentation-at-point
: If point is within a presentation, inspect the associated object with the SLIME inspector.

C-c C-v n
M-x slime-next-presentation
: Move point to the next presentation in the buffer.

C-c C-v p
M-x slime-previous-presentation
: Move point to the previous presentation in the buffer.

Similar operations are also possible from the context menu of every presentation. Using *mouse-3* on a presentation, the context menu opens and offers various commands. For some objects, specialized commands are also offered. Users can define additional specialized commands by defining a method for `swank::menu-choices-for-presentation`.

Warning: On Lisp implementations without weak hash tables, all objects associated with presentations are protected from garbage collection. If your Lisp image grows too large because of that, use *C-c C-v M-o* (`slime-clear-presentations`) to remove these associations. You can also use the command *C-c M-o* (`slime-repl-clear-buffer`), which both clears the REPL buffer and removes all associations of objects with presentations.

Warning: Presentations can confuse new users.

```
CL-USER> (cons 1 2)
(1 . 2)
CL-USER> (eq '(1 . 2) '(1 . 2))
T
```

One could have expected `NIL` here, because it looks like two fresh cons cells are compared regarding object identity. However, in the example the presentation *`(1 . 2)`* was copied twice to the REPL. Thus `EQ` is really invoked with the same object, namely the cons cell that was returned by the first form entered in the REPL.

8.13 Typeout frames

A "typeout frame" is a special Emacs frame which is used instead of the echo area (minibuffer) to display messages from SLIME commands. This is an optional feature. The advantage of a typeout frame over the echo area is that it can hold more text, it can be scrolled, and its contents don't disappear when you press a key. All potentially long messages are sent to the typeout frame, such as argument lists, macro expansions, and so on.

`M-x slime-ensure-typeout-frame`
: Ensure that a typeout frame exists, creating one if necessary.

If the typeout frame is closed then the echo area will be used again as usual.

To have a typeout frame created automatically at startup you should load the `slime-typeout-frame` package. (see Section 8.1 [Loading Contribs], page 28.)

The variable `slime-typeout-frame-properties` specifies the height and possibly other properties of the frame. Its value is passed to `make-frame`. (See Info file '`elisp`', node '`Creating Frames`'.)

8.14 TRAMP

The package `slime-tramp` provides some functions to set up filename translations for TRAMP. (see Section 7.1.3 [Setting up pathname translations], page 26)

8.15 Documentation Links

For certain error messages, SBCL includes references to the ANSI Standard or the SBCL User Manual. The `slime-references` package turns those references into clickable links. This makes finding the referenced section of the HyperSpec much easier.

8.16 Xref and Class Browser

A rudimentary class browser is provied by the `slime-xref-browser` package.

`M-x slime-browse-classes`
: This command asks for a class name and displays inheritance tree of for the class.

`M-x slime-browse-xrefs`
: This command prompts for a symbol and the kind of cross reference, e.g. callers. The cross reference tree rooted at the symbol is then then displayed.

8.17 Highlight Edits

`slime-highlight-edits` is a minor mode to highlight those regions in a Lisp source file which are modified. This is useful to quickly find those functions which need to be recompiled (whith *C-c C-c*)

M-x slime-highlight-edits-mode
: Turns `slime-highlight-edits-mode` on or off.

8.18 Scratch Buffer

The SLIME scratch buffer, in contrib package `slime-scratch`, imitates Emacs' usual `*scratch*` buffer. If `slime-scratch-file` is set, it is used to back the scratch buffer, making it persistent. The buffer is like any other Lisp buffer, except for the command bound to *C-j*.

C-j
M-x slime-eval-print-last-expression
: Evaluate the expression sexp before point and insert print value into the current buffer.

M-x slime-scratch
: Create a '`*slime-scratch*`' buffer. In this buffer you can enter Lisp expressions and evaluate them with *C-j*, like in Emacs's '`*scratch*`' buffer.

8.19 Meta package: `slime-fancy`

`slime-fancy` is a meta package which loads a combination of the most popular packages.

9 Credits

The soppy ending...

Hackers of the good hack

SLIME is an Extension of SLIM by Eric Marsden. At the time of writing, the authors and code-contributors of SLIME are:

Helmut Eller
Matthias Koeppe
Edi Weitz
Peter Seibel
Martin Simmons
Wolfgang Jenkner
Lawrence Mitchell
Brian Downing
Zach Beane
Antonio Menezes Leitao
Thomas F. Burdick
Matthew Danish
Andreas Fuchs
Raymond Toy
Joerg Hoehle
Travis Cross
Stelian Ionescu
Robert Lehr
Mark Harig
Ian Eslick
Eduardo Muñoz
Bjørn Nordbø
Alan Caulkins
Wojciech Kaczmarek
Tim Daly Jr.
Steve Smith
Rui Patrocínio
R. Matthew Emerson
Neil Van Dyke
Mikel Bancroft
Masayuki Onjo
Lynn Quam
Knut Olav Bøhmer
Juergen Gmeiner
Jan Rychter
Ivan Boldyrev
Gerd Flaig
Didier Verna
Luke Gorrie
Marco Baringer
Nikodemus Siivola
Christophe Rhodes
Douglas Crosher
Geo Carncross
Gábor Melis
Bill Clementson
Mark Evenson
Utz-Uwe Haus
Takehiko Abe
James Bielman
Willem Broekema
Lars Magne Ingebrigtsen
Bryan O'Connor
Tobias Rittweiler
Stefan Kamphausen
Robert E. Brown
Jouni K Seppanen
Gary King
Christian Lynbech
Ariel Badichi
Yaroslav Kavenchuk
William Bland
Sven Van Caekenberghe
Stas Boukarev
Robert Macomber
Pawel Ostrowski
NIIMI Satoshi
Matthew D. Swank
Mark Wooding
Levente Mészáros
Kai Kaminski
Jon Allen Boone
James McIlree
Ignas Mikalajunas
Frederic Brunel
David Reitter
Tobias C. Rittweiler
Alan Ruttenberg
Juho Snellman
Attila Lendvai
Daniel Barlow
Michael Weber
Luís Oliveira
Andras Simon
Espen Wiborg
Thomas Schilling
Richard M Kreuter
Harald Hanche-Olsen
Taylor R. Campbell
John Paul Wallington
Alan Shutko
Tiago Maduro-Dias
Sean O'Rourke
Nathan Bird
Ivan Toshkov
Eric Blood
Chris Capel
Alexey Dejneka
Wolfgang Mederle
Tom Pierce
Svein Ove Aas
Russell McManus
Reini Urban
Paul Collins
Mészáros Levente
Matt Pillsbury
Marco Monteiro
Lasse Rasinen
Julian Stecklina
Johan Bockgrd
Ivan Shvedunov
Hannu Koivisto
Dustin Long
Daniel Koning

Dan Weinreb	Dan Pierson	Brian Mastenbrook
Brandon Bergren	Bob Halley	Barry Fishman
B.Scott Michel	Anton Vodonosov	Aleksandar Bakic
Alain Picard		

... not counting the bundled code from '`hyperspec.el`', *CLOCC*, and the *CMU AI Repository*.

Many people on the `slime-devel` mailing list have made non-code contributions to SLIME. Life is hard though: you gotta send code to get your name in the manual. `:-)`

Thanks!

We're indebted to the good people of `common-lisp.net` for their hosting and help, and for rescuing us from "Sourceforge hell."

Implementors of the Lisps that we support have been a great help. We'd like to thank the CMUCL maintainers for their helpful answers, Craig Norvell and Kevin Layer at Franz providing Allegro CL licenses for SLIME development, and Peter Graves for his help to get SLIME running with ABCL.

Most of all we're happy to be working with the Lisp implementors who've joined in the SLIME development: Dan Barlow and Christophe Rhodes of SBCL, Gary Byers of OpenMCL, and Martin Simmons of LispWorks. Thanks also to Alain Picard and Memetrics for funding Martin's initial work on the LispWorks backend!

Key (Character) Index

N

O

P

Q

R

S

T

V

X

Command and Function Index

Variable and Concept Index