CHAPTER 18
SLIME

Introduction

The *Superior Lisp Interaction Mode for Emacs* (or *SLIME* for short) is a widely used, powerful programming environment for Lisp programs. If you're using Emacs to edit Common Lisp code then you really should consider working under SLIME. It integrates and standardizes the four basic Common Lisp development tools (listener, editor, inspector and debugger) and augments these with some handy introspective interfaces (such as definition locators and advanced symbol completion). Simple "key chords" control the most common actions: compiling a single definition or a whole file, inspecting values or locating source from backtraces, choosing a restart after an error. SLIME will boost your productivity; by making information more instantly accessible it leaves you free to get on with thinking about your application.

SLIME is open source and free to use. It supports ten out of the eleven Common Lisp implementations listed in Chapter 1, the exception being GNU Common Lisp (GCL); it runs under GNU Emacs versions 21-23 and XEmacs version 21; it works on Unix, Mac OS X, and Windows. We'll be working here with Clozure CL and GNU Emacs running on FreeBSD but could equally have chosen a different combination and very little would have changed.

The message of this chapter is that there's little excuse for debugging complex applications from a raw command line, and none for ignoring Lisp's introspection facilities altogether. We'll walk through SLIME's most useful features and discuss some of the hairier configuration issues. I have to assume familiarity with Emacs; the initial learning curve for this application is quite steep and it's probably beyond my remit to insist that you should study it just so you can follow this chapter. So if you're new to Emacs I suggest you read "Background" on page 2 below and then skim "Basic Operations" on page 4. That'll give you a flavor of the potential of SLIME and hence Common Lisp itself to support application development at a very high level. Even if you never use SLIME, you could choose to lift ideas from the backend files it uses for talking to a range of Lisp implementations.

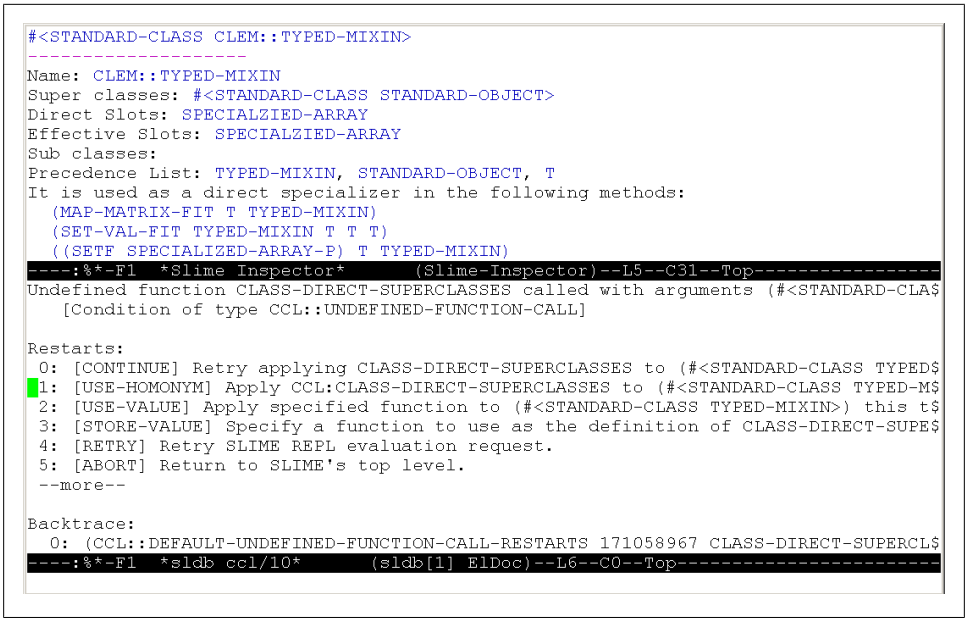


Figure 18-1. An Emacs window showing the SLIME inspector and debugger at work.

Background

Emacs is a freely available, open source text editor which runs in both terminal windows (Figure 18-1) and GUIs on a very wide variety of platforms. There are several ports, the most popular being *GNU Emacs*. You'll find a good general-purpose introduction on Wikipedia, much more detail in the O'Reilly book *Learning GNU Emacs* by Debra Cameron, Bill Rosenblatt and Eric Raymond, and an extensive project website at

http://www.gnu.org/software/emacs/

Exercise

If you've never used Emacs before, visit *http://www.gnu.org/software/emacs/tour/* and wander through the tour.

In Emacs documentation and in this chapter, `C-q` stands for `control-q` and `M-q` for `meta-q`. Most modern keyboards don't have a *Meta* key. Try `alt-q` (or `command-q` on Mac OS); if that fails then `Escape` followed by `q` is clunky but will work. `M-C-q` means `meta-control-q` (or `ESC control-q`).

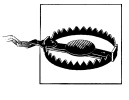
Much of Emacs is written in Lisp. Emacs can be extended by writing more Lisp. Every character you type (whether the plain letter "q" or some fancy control sequence) invokes a Lisp function; in the finest traditions of Lisp such functions can be defined and

redefined on the fly, i.e. without restarting Emacs. When it starts up, Emacs loads Lisp forms from the *.emacs* file (if you have one) in your home directory. Most long-standing Emacs users will have accumulated a number of personalizations in their *.emacs* over the years, whether forms to load various libraries (such as SLIME), or tweaks to the Emacs source, or functionality extensions. I'm using Emacs to write this book; here as an example of Lisp that's not Common Lisp is the part of my *.emacs* which I added to support this activity.

```
;; Control whether quotes are curly (“”) or straight ("")

(global-set-key [f9] 'toggle-sgml-quotes)

(define-key sgml-mode-map "'" 'sgml-insert-single-quote)
(define-key sgml-mode-map "\"" 'sgml-insert-double-quote)

(defvar sgml-quotes-p nil)

(defun toggle-sgml-quotes ()
  (interactive)
  (setf sgml-quotes-p (not sgml-quotes-p))
  (message (format "Fancy quotes %s" (if sgml-quotes-p "on" "off"))))

(defun sgml-insert-single-quote ()
  (interactive)
  (insert (if sgml-quotes-p "’" "'")))

(defun sgml-insert-double-quote (p)
  (interactive "P")
  (insert (if sgml-quotes-p
              (if p "”" "“")         ; C-u " for close quote
            "\"")))
```

The Lisp in Emacs, *Emacs Lisp* provides built-in support for programming the Emacs text editor with functions such as `global-set-key` in the example above. But the language is not Common Lisp. It's a lot smaller; there are some important differences (for example, all variables have dynamic scope) and a host of lesser ones (e.g. `format` is different, as you'll have noticed in `toggle-sgml-quotes`). You can read more about Emacs Lisp in the O'Reilly book mentioned above or by following links from the Emacs project website. You can extend Emacs Lisp to something closer to Common Lisp by evaluating the form

```
(require 'cl)
```



This still doesn't give you the whole of Common Lisp. In particular CLOS is absent. See *http://www.gnu.org/software/emacs/manual/html_mono/cl.html* for details.

Emacs is much more than an editor. Of particular interest here: you can run shells in Emacs windows; you might then run an interactive Lisp session in such a shell. Emacs supports this with *Inferior Lisp mode* (the word "inferior" here is not a value judgement

but means that Lisp is running as a subprogram under the auspices of Emacs). And then, for example, if you're editing Lisp code the key binding `M-C-x` sends the definition under the cursor to your inferior Lisp for evaluation. SLIME is a rich extension of the Inferior Lisp mode.

Emacs comes bundled with Linux, FreeBSD, and Mac OS X.

Exercise

Estimate how many people have access (whether or not they use it) to Emacs Lisp.

Emacs uses the term *point* to refer to the text cursor location and I'll be doing the same here. So "the symbol under point" means "the symbol under the text cursor".

Basic Operations

Let's Get Started

This should be very straightforward. The project home page for SLIME:

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

includes links for a thorough manual, an hour-long video tutorial, the slime-devel mailing list which should be your first port of call if you need technical help, and for downloads:

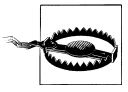
http://common-lisp.net/project/slime/snapshots/slime-current.tgz

Download and unpack the distribution, for example using `curl` and `tar` as described under "Download" in Chapter 17. Configure SLIME by adding forms to your *.emacs*, along the following lines:

```
(setq inferior-lisp-program "/home/ndl/lisps/ccl/scripts/ccl")
(add-to-list 'load-path "/home/ndl/chapter-18/slime-2009-08-26/")
(require 'slime-autoloads)
(slime-setup '(slime-fancy))
```

I really did hope to write this book without mentioning `setq`. Think of it as a prehistoric `setf` which only works on symbol values. If you `(require 'cl)` then Emacs will understand `setf`.

- `inferior-lisp-program` is the location of your Lisp executable. You can pass command-line arguments to this Lisp (for example, `"-tty"` to prevent LispWorks from opening its windowing GUI) by adding them to the end of this string.



If (typically on Windows) there are spaces in the executable's pathname and you're using XEmacs then you will run into difficulties with `inferior-lisp-program`; it can't tell where the filename ends and its arguments begin. use instead the configuration variable `slime-lisp-implementations` (see the SLIME manual for details) which does not suffer from this problem.

- The string to pass to `add-to-list` is the directory into which you unpacked SLIME.
- There are some additional features implied by the two last forms. We'll come back to these later, in "Configuration" on page 13. Note for now that specifying `slime-fancy` loads a number of optional modules, several of which are assumed later in this chapter.

Finally, evaluate the above forms (e.g. `M-x eval-region`) and start SLIME itself:

```
M-x slime
```

Problems? Check your *.emacs* configuration; make sure the paths you supplied are correct.

What Have We Got?

When SLIME starts, it fires up the inferior Lisp and loads into it a server called *Swank* with which it then communicates using a socket protocol. Swank comes with a backend file for every supported implementation.

Exercise

Locate and take a brief look at your implementation's backend file. They're all in the top level SLIME directory. Also take a look at *swank.asd* which includes instructions for driving Swank independently of Emacs.

SLIME creates three initial buffers, two of which are not much use other than for troubleshooting. `*inferior-lisp*` is your Lisp's top level; depending on platform it might contain a working REPL but if it does it will lack the power provided by the listener in the main REPL buffer (named `*slime-repl ccl*` on CCL, etc.) which you'll find topmost. `*slime-events*` contains a full log of communications between SLIME and the inferior Lisp.

Some SLIME features don't work with all Lisps. This will depend on what the underlying implementation supports. Look for "Warning: These Swank interfaces are unimplemented" in the `*inferior-lisp*` buffer. (We'll restrict ourselves here to features with universal support.)

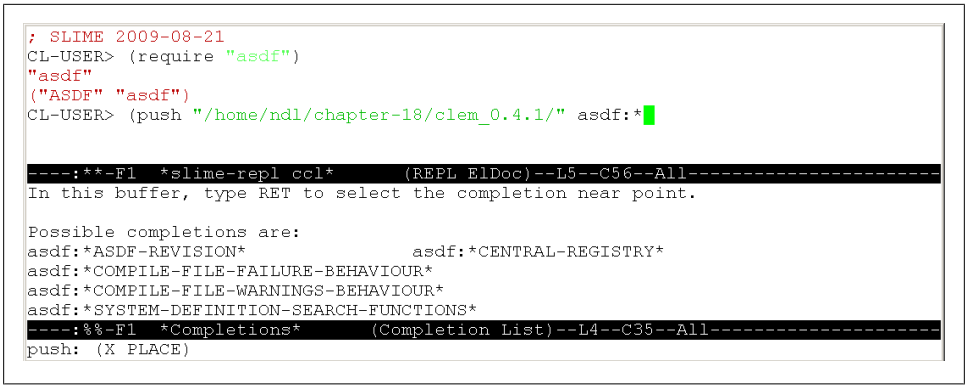


*Figure 18-2. Interacting with the listener. The echo area shows the argument list for the form we're entering and the *Completions* buffer lists all external symbols in the ASDF package starting with the character **.

Useful Key Bindings

In both source files and the listener, `M-TAB` completes the symbol under point. `M-C-i` is an equivalent key binding which you may find more natural to work with (and on Windows and Mac OS it won't be intercepted by the window manager!). In the listener `TAB` on its own has a dual action: it attempts to indent but if that doesn't change anything then it performs symbol completion instead. So `TAB TAB` is generally a quick and easy way to fix listener indentation and complete the current symbol.

SLIME's completion can expand each component of a hyphenated symbol: `m-v-b` completes to `multiple-value-bind` and `h--t` to `hash-table-test`. It also provides context-sensitive completion for keywords: `(make-array 3 :e<TAB>` completes the keyword to `:element-type` (because no other keywords starting with `:e` are valid here).

Symbol completion assumes and encourages consistent use of lower case.

The space key inserts a space character but has a secondary action in both source files and the listener: it prints to the echo area the argument list of the operator which you're working on, highlighting the argument which comes next (Figure 18-2). If you want to do this without inserting spaces (for example, in a read-only buffer) then use `M-x slime-arglist`.

You can look up *Hyperspec* documentation for the symbol under point with `C-c C-d h` and (an occasional lifesaver) for the format character under point with `C-c C-d ~`. Both of these will open a web browser using the function named by the Emacs variable `browse-url-browser-function`. If you have a local copy of the Hyperspec then reset `common-lisp-hyperspec-root`, along the lines of `"file:/usr/local/doc/HyperSpec/"`.

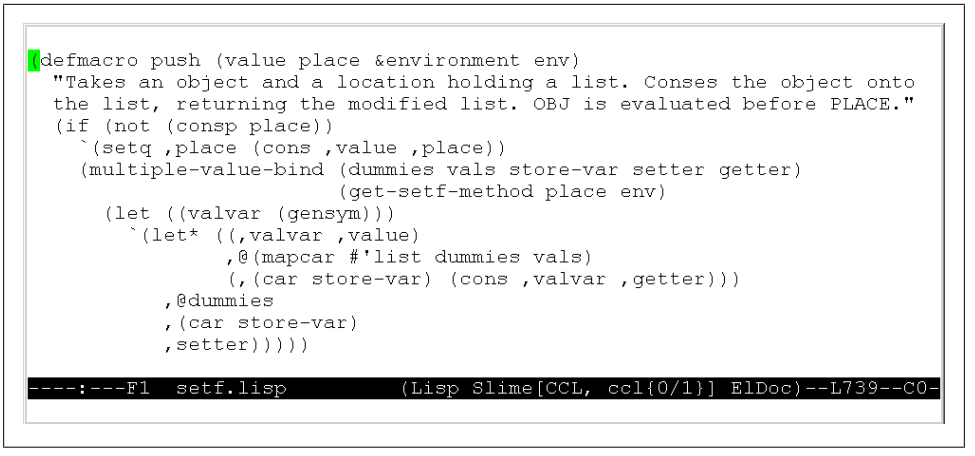


Figure 18-3. Using meta-. to explore an open source Lisp implementation (Clozure CL).

The `slime-selector` command is a shortcut for switching buffers. It doesn't have a default key binding but you can fix that, by adding this or similar to your *.emacs*:

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

It offers you a choice of single letter "commands" for specifying which buffer you're after, among which are:

- `r` for the REPL buffer (i.e. the listener),
- `d` for the debugger (which we'll introduce in "SLDB: The Debugger" on page 9 below),
- `l` for the most recently visited source buffer,
- `?` for a full list of options.

The upshot of this is that `C-s s r` will take you back to the REPL.

If you accidentally kill the REPL buffer, open a new listener with `M-x slime-switch-to-output-buffer`. In other SLIME buffers (e.g. source files), `C-c C-z` is bound to this.

Locating Source

`M-.` locates source code: it takes you to the definition of the symbol under point. The information for this (which definition was loaded from where) comes from your inferior Lisp. Unless you've configured Lisp not to store definition locations (see Chapter 28 for more on this) `M-.` should always work on your own code. If you're using one of the open source Lisps, it might also give you easy access to underlying implementation details.

Exercise

Pick a Common Lisp symbol with a straightforward definition (`when` might be a good place to start), type it into the listener and see whether `M-.` gets you anywhere. If that succeeded, try again with something more interesting (Figure 18-3). Try not to get sucked in.

How well `M-.` works is implementation-dependent; typically it'll do better with compiled than interpreted code. (See Chapter 12 for more about this distinction.)

Note while you're floating around the source that SLIME understands reader conditionals: `#+not-this-one` will "grey out" a code block in the same way that semicolons do to a single line.

If you've used `M-.` then `M-,` will take you back to where you were beforehand. `M-.` pushes locations onto a stack and `M-,` pops it.

The command `C-x 4 .` shows source in the "other" window, allowing you to view caller and callee together.

Once you've found the function you're looking for, maybe you'd like to trace it? Use `C-c C-t` to toggle whether the symbol at point is traced.

Compiling source

The workhorse for compiling and executing the top-level form at point is `C-c C-c`. You can control optimizations by supplying a numeric prefix argument: positive for maximal debuggability, negative to compile for speed.

To compile and load the current source file, use `C-c C-k`.

If the compiler hits badly broken syntax, such as `(if)`, then an error will be signaled and you'll be thrown into the debugger. See "SLDB: The Debugger" on page 9 below for how to get out that.

If the compiler issues any notes or warnings then the offending forms will be annotated (by underlining them) and the warnings listed in a `*SLIME Compilation*` buffer (Figure 18-4). If you're running Emacs in a GUI then place the mouse over an annotation to see its message as a popup. Otherwise, cycle through the messages using `M-n` to go to the next note and `M-p` to back up. When you're done, clear the annotations from the buffer with `C-c M-c`.

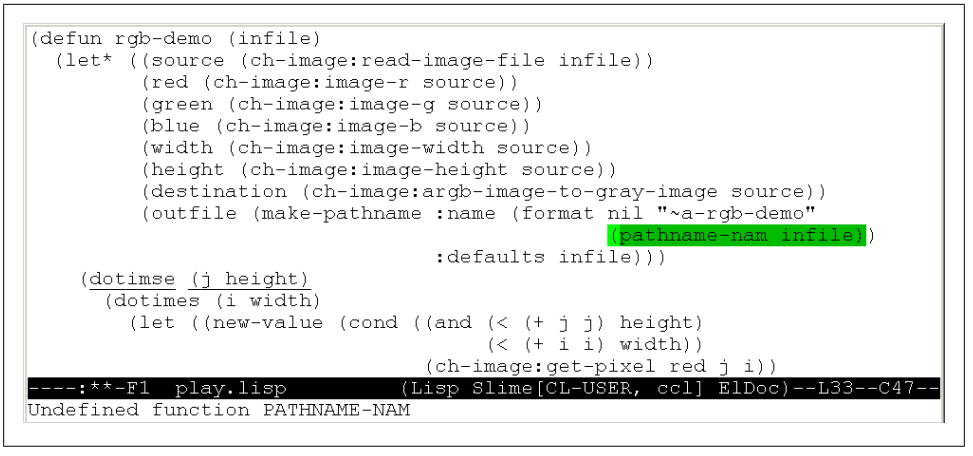
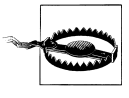


Figure 18-4. Compiler warnings provoked by two deliberate mistakes. The source for all warnings is underlined and the "current" one highlighted, a description of the warning itself is in the echo area. Use M-n (next) and M-p (previous) to cycle through the warnings.

SLIME also provides commands for evaluating without explicit invocation of the compiler. In particular, `M-C-x` evaluates the current top-level form. Note that if the form is a `defvar` then the variable concerned will be reset even if it already had a value (as if the form were a `defparameter`).

Exercise

We've seen the `M-C-x` binding before. Where?



`C-c C-c` writes the form to a temporary file and gets Lisp to `compile-file` that; `M-C-x` injects the form directly into the inferior Lisp. In the first case `*load-pathname*` will point to the temporary file and in the second it won't be set at all. If you need the correct value of `*load-pathname*`, compile the whole file (`C-c C-k`).

SLDB: The Debugger

Any errors signaled by the inferior Lisp are debugged in a popup *SLDB* buffer (Figure 18-5). SLDB supplies a number of commands for navigating through stack frames and invoking restarts. You don't have to remember what all of these are because `RET` will (to quote the SLIME manual) "do the most obvious useful thing".

- If point is over a restart and you press `RET` then that restart will be invoked.
- Alternatively: `c` invokes `continue`, `a` is for `abort`, the digits `0` through `9` invoke numbered restarts, and `q` quits the debugger altogether.

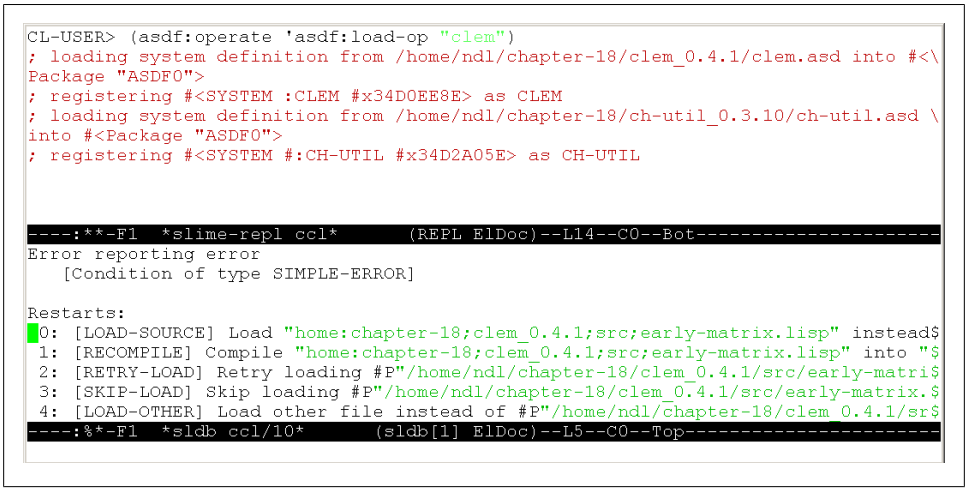


Figure 18-5. Entering the debugger. The condition is reported first, followed by the restarts. A backtrace follows further down.

- If point is over a frame in the backtrace then `RET` toggles whether that frame's variables are displayed.
- You can use `v` to see the source code associated with that frame; your position in the source will be highlighted.
- Combine these actions with `M-n` and `M-p` for moving down or up the stack respectively, hiding the old frame and displaying both the new frame's variables and your location in its source (Figure 18-6).
- Invoke the disassembler with `D`.
- If point is over a frame variable then `RET` will inspect its value. `i` will prompt for a value to inspect (defaulting to the value at point); you can refer to variables in the current frame in the expression you supply.
- `e` prompts for an expression and evaluates it; again you can refer to local variables.

Exercise

Evaluate something which will signal an error. If you don't have anything else to hand then `(defun foo (I-have-a-value) I-do-not)` will do fine. Alternatively, add a `(break)` to some working code, for example ch-image. Gain familiarity with SLDB by trying out all of the above.

To interrupt Lisp and enter the debugger, type `C-c C-b`.

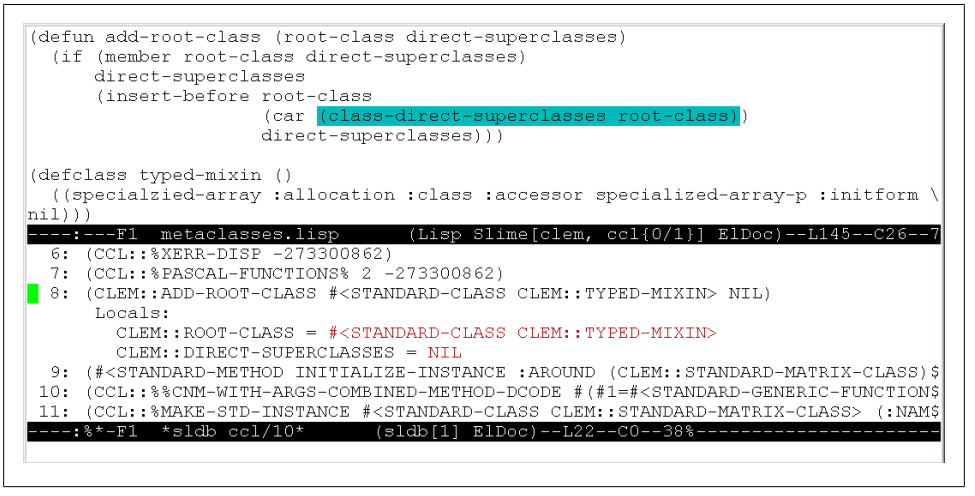


Figure 18-6. Frame details in SLDB. M-n shows the variables from the next frame down and—if it can—also opens the source file for that frame and highlights your position in the source.

The Inspector

You need to know what's going on inside your data structures; poking around in the guts of objects which you've grabbed from the listener or debugger should become second nature. This is where the *inspector*, one of the most powerful tools available to Lisp programmers, comes in. This is a tool for displaying a "snapshot" of some Lisp object, exposing its components and allowing you to reset them or inspect them recursively.

We've already met the SLDB shortcuts for invoking the inspector. If you're in another SLIME buffer use `C-c I`. Like `i` in the debugger this prompts for a value to inspect, defaulting to the value at point.

Inspect the last value printed by the REPL with `C-c I`; when prompted for a value enter *.

The inspector's display varies according to the class of the object you're inspecting. The convention is to show a human-readable description at the top followed by a list of internal values (methods for a generic function, slots for a general CLOS object, and so on).

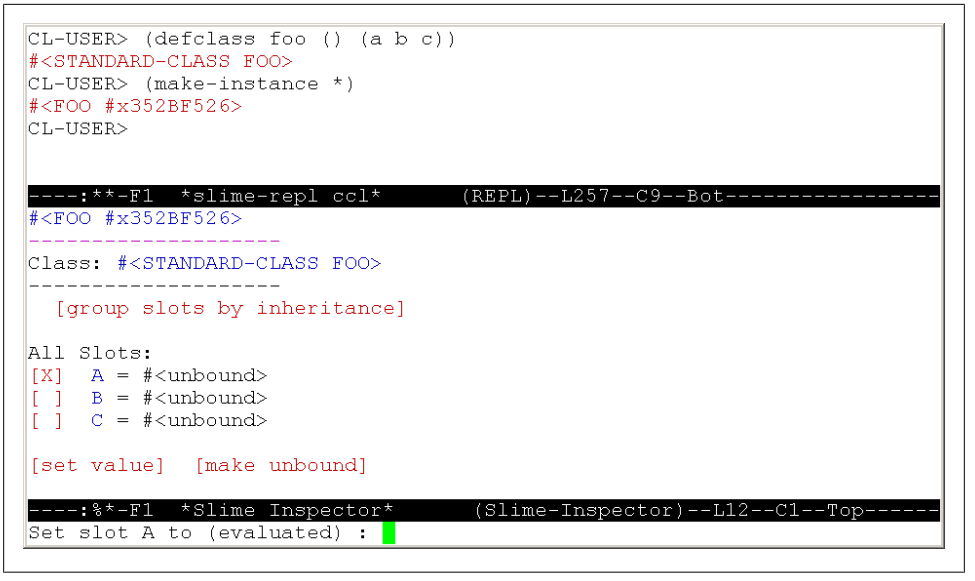
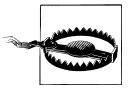


Figure 18-7. Inspecting a simple CLOS object. See the exercise in this section.

You can control how this display works; see "Customizing the Inspector" on page 16 below.

Once you're in the inspector, as for the debugger, `RET` acts as a context-sensitive operator. If point is on a value then `RET` will inspect that instead; if point is on an action—these are displayed in bright red and within square brackets—then either `RET` or a mouse click will invoke it. Actions are themselves context-sensitive and include removing function definitions from symbols, setting or unbinding slot values, clearing hash tables.



Take care not to accidentally invoke actions which will destroy your data. A one-key action to clear a hash table isn't always that convenient. To override the supplied behavior, see "Customizing the Inspector" on page 16 below.

SLDB maintains a stack of inspected values: each time you press `RET` to inspect something, a new inspected value gets added to the stack (and shown to you). You can navigate back to previous inspected values with `l`. Use either `n` or `SPC` to go the other way and retrace your steps further in.

Exercise

(Figure 18-7) Define a class and inspect an instance of it; use `RET` first to mark one of its slots and then to set a value into that slot. Use `RET` again to inspect the new value. Use `M-RET` to set the value under point into `*`; experiment with `n` and `l`. Now quit the inspector (the key binding `q` works here as it did for the debugger) and verify in the REPL what's happened to * and **, and that your instance has been updated.

Two other useful inspector key bindings are:

- `.` to show you the source code associated with the value under point, and
- `g` to update the inspector display and present an updated snapshot of the object you're looking at: use this if the object has been modified elsewhere since you opened this view.

Advanced Use

Configuration

We saw earlier a basic configuration technique for SLIME: you register the location of an inferior Lisp, tell Emacs where to find SLIME, and then call:

```
(require 'slime-autoloads)
(slime-setup '(slime-fancy))
```

The `require` form above means that SLIME will be loaded "on demand" when you need it. If you'd rather always load SLIME when Emacs is started, replace that form with `(require 'slime)`. If you'd like to guarantee that SLIME is running whenever you open a Lisp file, add the following to your *.emacs* (this will work with either require form):

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

The argument to `slime-setup` is an optional list of "contributed modules" to be loaded alongside SLIME itself. To load SLIME on its own you'd replace the call with `(slime-setup)`. Not all of the contributed modules are documented in the manual; the best way to see what's on offer is to look through SLIME's *contrib/* subdirectory. Specifying `slime-fancy` gives you a usable set of about half of the available modules (see *contrib/slime-fancy.el*). As noted before, this chapter is based on the assumption that `slime-fancy` has been loaded.

You're not confined to always using the same inferior Lisp. If you invoke `M-x slime` with a prefix argument you'll be prompted for the Lisp to run. If the prefix is negative (`M-- M-x slime`) you'll be offered a choice based on the value of `slime-lisp-implemen`

tations; you'll find this variable described in the SLIME manual in the *Setup Tuning* section, under *Multiple Lisps*.

In addition to the configurations above, both SLIME and Swank can be *customized*. For SLIME, use the Emacs facility `M-x customize-group slime`. The available options are documented within the customization system (sometimes rather tersely). You might want to start by considering `slime-truncate-lines` (in the `Slime Ui` group). This is on by default and means that line wrapping is disabled in popup buffers (such as backtraces).

Swank is customized by setting variable values in your *~/.swank* file. The available options are documented in the SLIME manual. Take note of `swank:*communication-style*` which governs the interaction between Emacs and Swank. Each backend defines an appropriate value for this variable; for preference this should be `:spawn` (meaning: execute each request in a separate thread).

Support for Macros

You can invoke macroexpansion of the form beginning or ending at point with any of:

- `C-c C-m` to `macroexpand-1` the form,
- `C-u C-c C-m` to `macroexpand` the form (repeated calls to `macroexpand-1`), and
- `C-c M-m` to *walk* the form, fully macroexpanding the form and all its subforms.

The macroexpansion appears in a popup buffer (so you can close it with `q`). Further macroexpansions from the popup buffer modify its contents rather than opening further buffers.

SLIME manages indentation of macro bodies (for example: in a macroexpansion buffer, or in the listener) by treating `&body` arguments specially. Usually this works out fine. But if you happen to have two macros named with the same `symbol-name` (in different packages) then SLIME will arbitrarily choose the argument list of one to indent the other. List such collisions by calling `(swank:print-indentation-lossage)` and fix them on the Emacs side by setting the losing symbol's `common-lisp-indent-function` property.

Discover suitable values for this property either from the symbol you care about in an Emacs session in which that symbol hasn't been overridden yet, or by looking at the properties of other symbols that have similar argument lists.

SLIME caches symbols' indentation properties. It will stay up-to-date on macro definitions in the current package. When a new package is created (typically this means you've been loading code) SLIME scans every symbol in the system. These strategies

are usually good enough and in the rare cases when they're not you can force an update with `M-x slime-update-indentation`.

Remote Connections

The separation of SLIME into an Emacs interface and the Swank backend makes it relatively easy to run the Emacs side and Lisp session on different machines. This practice might come in handy if you ever need to debug a remote server. You'll need ssh access to the server's host and you might want to perform the first of the following steps before the server runs aground.

Terminology: the machine running code which you want to debug is the "server" (in particular it's going to server Swamk) and is "remote" to you. Your machine is the "client".

1. Configure the remote Lisp to act as a Swank server.
   - Load *swank-loader.lisp* from the SLIME distribution.
   - Evaluate these forms:

     ```
     (swank-loader:init)
     (swank:create-server :dont-close t)
     ```

   This will listen for connections on port 4005.

   Setting `dont-close` means the server won't stop serving Swank after the first connection goes away. That's usually what you want.
2. Open an ssh tunnel on port 4005 from client (your localhost, IP 127.0.0.1) to server. From a Unix command line:

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

   If you're on Windows, use *PuTTY* from *http://www.chiark.greenend.org.uk/~sgtatham/putty/* to open a tunnel.
3. Connect to the client's end of the tunnel: `M-x slime-connect` and accept both the default host (127.0.0.1) and the default port number (4005).

If you want to serve Swank on a port other than 4005 then specify a value for `:port` in the call to `swank:create-server` and change the second number in the ssh setup accordingly. For example: `(swank:create-server :port 84005 :dont-close t)` and `ssh -L4005:127.0.0.1:84005 username@remote.example.com`. (You can also switch the local port number by changing the first number in the ssh setup and overriding the value offered by the `slime-connect` command.)

If you want to share source files between client and server, you can do this using *TRAMP* (remote file editing module for GNU Emacs). Load the `slime-tramp` module from SLIME's *"contrib/"* directory into your local Emacs session and check out the section "Setting up pathname translations" in the SLIME manual.

Customizing the Inspector

The inspector display is controlled by the generic function `swank-backend:emacs-inspect`. Several methods will be defined on this already. You should feel free to raid them as templates for adding more if you need to specialize the display.

Use `M-. swank-backend:emacs-inspect` to list existing methods.

Each method takes an object as its single argument and should return a list specifying how to render the object for inspection. Every element of the list must be one of the following forms:

- A string will be inserted into the buffer as is.
- `(:value object &optional string)` renders an inspectable object (typically a slot or attribute of the object we're inspecting already). If `string` is provided it will be rendered in place of the value, otherwise `princ-to-string` is used to generate the display.
- `(:newline)` renders a newline.
- `(:action label lambda &key (refresh t))` - Render label (a text string) which when clicked will invoke `lambda`. If `refresh` is non-`nil` the currently inspected object will be re-inspected after calling the lambda.

Here's a very simple example:

```
(defclass animal ()
  ((legs :accessor animal-leg-count :initform 0)))

(defmethod swank-backend:emacs-inspect ((self animal))
  (let ((count (animal-leg-count self)))
    `("Hello. "
      (:value ,count ,(format nil "This animal has ~r leg~:p." count))
      (:newline) (:newline)
      (:action "Grow another one"
               ,(lambda () (incf (animal-leg-count self)))))))
```

A new instance of `animal` will look like this in the inspector:

```
#<ANIMAL 200CE94F>
--------------------
Hello. This animal has zero legs.
```

```
Grow another one
```

Select the first line of text to inspect the number zero. Click on the action to increment the animal's legs slot and update the display.

Summary of Key Bindings

Table 18-1. Key bindings introduced in this chapter

Data entry	
M-C-i	Complete symbol
TAB	Either indent or complete symbol (listener only)
Macro expansion	
C-c C-m	`macroexpand-1` form
C-u C-c C-m	Fully `macroexpand` form
C-c M-m	Walk form
q	Quit macroexpansion buffer
Documentation	
Space	Display argument list (in listener and source files)
C-c C-d h	Look up Hyperspec documentation
C-c C-d ~	Hyperspec documentation for format character
Navigation	
C-c s	Suggested binding for SLIME selector
M-.	Locate source
C-x 4 .	Locate source in other window
M-,	Return from source location
C-c C-t	Toggle whether definition is traced
Compilation	
C-c C-c	Compile this form
C-c C-k	Compile file
M-n and M-p	Next compiler annotation / previous annotation
C-c M-c and M-p	Clear annotations
M-C-x	Evaluate this form (re-evaluate `defvars`)
Debugger	
C-c C-b	Interrupt Lisp and enter debugger
RET	All-purpose context-sensitive action
M-n and M-p	Traverse stack: display values and show source

Debugger	
c	Continue
a	Abort
q	Quit debugger
v	Locate source
D	Disassemble source
e	Evaluate prompted form in lexical context of current frame
i	Inspect prompted form (evaluated in lexical context of current frame)
Inspector	
C-s I	Inspect prompted form
RET	All-purpose context-sensitive action
g	Refresh display
q	Quit inspector
M-RET	Set value under point to *
.	Locate source