

2. Lather, Rinse, Repeat: A Tour of the REPL

In this chapter you'll set up your programming environment and write your first Common Lisp programs. We'll use the easy-to-install Lisp in a Box developed by Matthew Danish and Mikel Evins, which packages a Common Lisp implementation with Emacs, a powerful Lisp-aware text editor, and SLIME,¹ a Common Lisp development environment built on top of Emacs.

This combo provides a state-of-the-art Common Lisp development environment that supports the incremental, interactive development style that characterizes Lisp programming. The SLIME environment has the added advantage of providing a fairly uniform user interface regardless of the operating system and Common Lisp implementation you choose. I'll use the Lisp in a Box environment in order to have a specific development environment to talk about; folks who want to explore other development environments such as the graphical integrated development environments (IDEs) provided by some of the commercial Lisp vendors or environments based on other editors shouldn't have too much trouble translating the basics.²

Choosing a Lisp Implementation

The first thing you have to do is to choose a Lisp implementation. This may seem like a strange thing to have to do for folks used to languages such as Perl, Python, Visual Basic (VB), C#, and Java. The difference between Common Lisp and these languages is that Common Lisp is defined by its standard--there is neither a single implementation controlled by a benevolent dictator, as with Perl and Python, nor a canonical implementation controlled by a single company, as with VB, C#, and Java. Anyone who wants to read the standard and implement the language is free to do so. Furthermore, changes to the standard have to be made in accordance with a process controlled by the standards body American National Standards Institute (ANSI). That process is designed to keep any one entity, such as a single vendor, from being able to arbitrarily change the standard.³ Thus, the Common Lisp standard is a contract between any Common Lisp vendor and Common Lisp programmers. The contract tells you that if you write a program that uses the features of the language the way they're described in the standard, you can count on your program behaving the same in any conforming implementation.

On the other hand, the standard may not cover everything you may want to do in your programs--some things were intentionally left unspecified in order to allow continuing experimentation by implementers in areas where there wasn't consensus about the best way for the language to support certain features. So every implementation offers some features above and beyond what's specified in the standard. Depending on what kind of programming you're going to be doing, it may make sense to just pick one implementation that has the extra features you need and use that. On the other hand, if we're delivering Lisp source to be used by others, such as libraries, you'll want--as far as possible--to write portable Common Lisp. For writing code that should be mostly portable but that needs facilities not defined by the standard, Common Lisp provides a

flexible way to write code "conditionalized" on the features available in a particular implementation. You'll see an example of this kind of code in Chapter 15 when we develop a simple library that smoothes over some differences between how different Lisp implementations deal with filenames.

For the moment, however, the most important characteristic of an implementation is whether it runs on our favorite operating system. The folks at Franz, makers of Allegro Common Lisp, are making available a trial version of their product for use with this book that runs on Linux, Windows, and OS X. Folks looking for an open-source implementation have several options. SBCL⁴ is a high-quality open-source implementation that compiles to native code and runs on a wide variety of Unices, including Linux and OS X. SBCL is derived from CMUCL,⁵ which is a Common Lisp developed at Carnegie Mellon University, and, like CMUCL, is largely in the public domain, except a few sections licensed under Berkeley Software Distribution (BSD) style licenses. CMUCL itself is another fine choice, though SBCL tends to be easier to install and now supports 21-bit Unicode.⁶ For OS X users, OpenMCL is an excellent choice--it compiles to machine code, supports threads, and has quite good integration with OS X's Carbon and Cocoa toolkits. Other open-source and commercial implementations are available. See Chapter 32 for resources from which you can get more information.

All the Lisp code in this book should work in any conforming Common Lisp implementation unless otherwise noted, and SLIME will smooth out some of the differences between implementations by providing us with a common interface for interacting with Lisp. The output shown in this book is from Allegro running on GNU/Linux; in some cases, other Lisp's may generate slightly different error messages or debugger output.

Getting Up and Running with Lisp in a Box

Since the Lisp in a Box packaging is designed to get new Lispers up and running in a first-rate Lisp development environment with minimum hassle, all you need to do to get it running is to grab the appropriate package for your operating system and the preferred Lisp from the Lisp in a Box Web site listed in Chapter 32 and then follow the installation instructions.

Since Lisp in a Box uses Emacs as its editor, you'll need to know at least a bit about how to use it. Perhaps the best way to get started with Emacs is to work through its built-in tutorial. To start the tutorial, select the first item of the Help menu, Emacs tutorial. Or press the Ctrl key, type `h`, release the Ctrl key, and then press `t`. Most Emacs commands are accessible via such key combinations; because key combinations are so common, Emacs users have a notation for describing key combinations that avoids having to constantly write out combinations such as "Press the Ctrl key, type `h`, release the Ctrl key, and then press `t`." Keys to be pressed together--a so-called key chord--are written together and separated by a hyphen. Keys, or key chords, to be pressed in sequence are separated by spaces. In a key chord, `C` represents the Ctrl key and `M` represents the Meta key (also known as Alt). Thus, we could write the key combination we just described that starts the tutorial like so: `C-h t`.

The tutorial describes other useful commands and the key combinations that invoke them. Emacs also comes with extensive online documentation using its own built-in hypertext documentation browser, Info. To read the manual, type `C-h i`. The Info system comes with its own tutorial, accessible simply by pressing `h` while reading the manual. Finally, Emacs provides quite a few

ways to get help, all bound to key combos starting with C-h. Typing C-h ? brings up a complete list. Two of the most useful, besides the tutorial, are C-h k, which lets us type any key combo and tells us what command it invokes, and C-h w, which lets us enter the name of a command and tells us what key combination invokes it.

The other crucial bit of Emacs terminology, for folks who refuse to work through the tutorial, is the notion of a *buffer*. While working in Emacs, each file you edit will be represented by a different buffer, only one of which is "current" at any given time. The current buffer receives all input--whatever you type and any commands you invoke. Buffers are also used to represent interactions with programs such as Common Lisp. Thus, one common action you'll take is to "switch buffers," which means to make a different buffer the current buffer so you can edit a particular file or interact with a particular program. The command `switch-to-buffer`, bound to the key combination C-x b, prompts for the name of a buffer in the area at the bottom of the Emacs frame. When entering a buffer name, hitting Tab will complete the name based on the characters typed so far or will show a list of possible completions. The prompt also suggests a default buffer, which you can accept just by hitting Return. You can also switch buffers by selecting a buffer from the Buffers menu.

In certain contexts, other key combinations may be available for switching to certain buffers. For instance, when editing Lisp source files, the key combo C-c C-z switches to the buffer where you interact with Lisp.

Free Your Mind: Interactive Programming

When you start Lisp in a Box, you should see a buffer containing a prompt that looks like this:

```
CL-USER>
```

This is the Lisp prompt. Like a Unix or DOS shell prompt, the Lisp prompt is a place where you can type expressions that will cause things to happen. However, instead of reading and interpreting a line of shell commands, Lisp reads Lisp expressions, evaluates them according to the rules of Lisp, and prints the result. Then it does it again with the next expression you type. That endless cycle of reading, evaluating, and printing is why it's called the *read-eval-print loop*, or REPL for short. It's also referred to as the *top-level*, the *top-level listener*, or the *Lisp listener*.

From within the environment provided by the REPL, you can define and redefine program elements such as variables, functions, classes, and methods; evaluate any Lisp expression; load files containing Lisp source code or compiled code; compile whole files or individual functions; enter the debugger; step through code; and inspect the state of individual Lisp objects.

All those facilities are built into the language, accessible via functions defined in the language standard. If you had to, you could build a pretty reasonable programming environment out of just the REPL and any text editor that knows how to properly indent Lisp code. But for the true Lisp programming experience, you need an environment, such as SLIME, that lets you interact with Lisp both via the REPL and while editing source files. For instance, you don't want to have to cut and paste a function definition from a source file to the REPL or have to load a whole file just because you changed one function; your Lisp environment should let us evaluate or compile both individual expressions and whole files directly from your editor.

Experimenting in the REPL

To try the REPL, you need a Lisp expression that can be read, evaluated, and printed. One of the simplest kinds of Lisp expressions is a number. At the Lisp prompt, you can type `10` followed by Return and should see something like this:

```
CL-USER> 10
10
```

The first `10` is the one you typed. The Lisp reader, the *R* in REPL, reads the text "`10`" and creates a Lisp object representing the number 10. This object is a *self-evaluating* object, which means that when given to the evaluator, the *E* in REPL, it evaluates to itself. This value is then given to the printer, which prints the `10` on the line by itself. While that may seem like a lot of work just to get back to where you started, things get a bit more interesting when you give Lisp something meatier to chew on. For instance, you can type `(+ 2 3)` at the Lisp prompt.

```
CL-USER> (+ 2 3)
5
```

Anything in parentheses is a list, in this case a list of three elements, the symbol `+`, and the numbers 2 and 3. Lisp, in general, evaluates lists by treating the first element as the name of a function and the rest of the elements as expressions to be evaluated to yield the arguments to the function. In this case, the symbol `+` names a function that performs addition. 2 and 3 evaluate to themselves and are then passed to the addition function, which returns 5. The value 5 is passed to the printer, which prints it. Lisp can evaluate a list expression in other ways, but we needn't get into them right away. First we have to write. . .


"Hello, World," Lisp Style

No programming book is complete without a "hello, world"⁷ program. As it turns out, it's trivially easy to get the REPL to print "hello, world."

```
CL-USER> "hello, world"
"hello, world"
```

This works because strings, like numbers, have a literal syntax that's understood by the Lisp reader and are self-evaluating objects: Lisp reads the double-quoted string and instantiates a string object in memory that, when evaluated, evaluates to itself and is then printed in the same literal syntax. The quotation marks aren't part of the string object in memory--they're just the syntax that tells the reader to read a string. The printer puts them back on when it prints the string because it tries to print objects in the same syntax the reader understands.

However, this may not really qualify as a "hello, world" *program*. It's more like the "hello, world" *value*.

You can take a step toward a real program by writing some code that as a side effect prints the string "hello, world" to standard output. Common Lisp provides a couple ways to emit output, but the most flexible is the **FORMAT** function. **FORMAT** takes a variable number of arguments, but the only two required arguments are the place to send the output and a string. You'll see in the next chapter how the string can contain embedded directives that allow you to interpolate subsequent arguments into the string,  la `printf` or Python's string-`%`. As long as the string

doesn't contain an ~, it will be emitted as-is. If you pass `t` as its first argument, it sends its output to standard output. So a **FORMAT** expression that will print "hello, world" looks like this:⁸

```
CL-USER> (format t "hello, world")
hello, world
NIL
```

One thing to note about the result of the **FORMAT** expression is the **NIL** on the line after the "hello, world" output. That **NIL** is the result of evaluating the **FORMAT** expression, printed by the REPL. (**NIL** is Lisp's version of false and/or null. More on that in Chapter 4.) Unlike the other expressions we've seen so far, a **FORMAT** expression is more interesting for its side effect--printing to standard output in this case--than for its return value. But every expression in Lisp evaluates to some result.⁹

However, it's still arguable whether you've yet written a true "program." But you're getting there. And you're seeing the bottom-up style of programming supported by the REPL: you can experiment with different approaches and build a solution from parts you've already tested. Now that you have a simple expression that does what you want, you just need to package it in a function. Functions are one of the basic program building blocks in Lisp and can be defined with a **DEFUN** expression such as this:

```
CL-USER> (defun hello-world () (format t "hello, world"))
HELLO-WORLD
```

The `hello-world` after the **DEFUN** is the name of the function. In Chapter 4 we'll look at exactly what characters can be used in a name, but for now suffice it to say that lots of characters, such as `-`, that are illegal in names in other languages are legal in Common Lisp. It's standard Lisp style--not to mention more in line with normal English typography--to form compound names with hyphens, such as `hello-world`, rather than with underscores, as in `hello_world`, or with inner caps such as `helloWorld`. The `()`s after the name delimit the parameter list, which is empty in this case because the function takes no arguments. The rest is the body of the function.

At one level, this expression, like all the others you've seen, is just another expression to be read, evaluated, and printed by the REPL. The return value in this case is the name of the function you just defined.¹⁰ But like the **FORMAT** expression, this expression is more interesting for the side effects it has than for its return value. Unlike the **FORMAT** expression, however, the side effects are invisible: when this expression is evaluated, a new function that takes no arguments and with the body `(format t "hello, world")` is created and given the name `HELLO-WORLD`.

Once you've defined the function, you can call it like this:

```
CL-USER> (hello-world)
hello, world
NIL
```

You can see that the output is just the same as when you evaluated the **FORMAT** expression directly, including the **NIL** value printed by the REPL. Functions in Common Lisp automatically return the value of the last expression evaluated.

Saving Your Work

You could argue that this is a complete "hello, world" program of sorts. However, it still has a problem. If you exit Lisp and restart, the function definition will be gone. Having written such a fine function, you'll want to save your work.

Easy enough. You just need to create a file in which to save the definition. In Emacs you can create a new file by typing `C-x C-f` and then, when Emacs prompts you, entering the name of the file you want to create. It doesn't matter particularly where you put the file. It's customary to name Common Lisp source files with a `.lisp` extension, though some folks use `.cl` instead.

Once you've created the file, you can type the definition you previously entered at the REPL. Some things to note are that after you type the opening parenthesis and the word **DEFUN**, at the bottom of the Emacs window, SLIME will tell you the arguments expected. The exact form will depend somewhat on what Common Lisp implementation you're using, but it'll probably look something like this:

```
(defun name varlist &rest body)
```

The message will disappear as you start to type each new element but will reappear each time you enter a space. When you're entering the definition in the file, you might choose to break the definition across two lines after the parameter list. If you hit Return and then Tab, SLIME will automatically indent the second line appropriately, like this:¹¹

```
(defun hello-world ()  
  (format t "hello, world"))
```

SLIME will also help match up the parentheses--as you type a closing parenthesis, it will flash the corresponding opening parenthesis. Or you can just type `C-c C-q` to invoke the command `slime-close-parens-at-point`, which will insert as many closing parentheses as necessary to match all the currently open parentheses.

Now you can get this definition into your Lisp environment in several ways. The easiest is to type `C-c C-c` with the cursor anywhere in or immediately after the **DEFUN** form, which runs the command `slime-compile-defun`, which in turn sends the definition to Lisp to be evaluated and compiled. To make sure this is working, you can make some change to `hello-world`, recompile it, and then go back to the REPL, using `C-c C-z` or `C-x b`, and call it again. For instance, you could make it a bit more grammatical.

```
(defun hello-world ()  
  (format t "Hello, world!"))
```

Next, recompile with `C-c C-c` and then type `C-c C-z` to switch to the REPL to try the new version.

```
CL-USER> (hello-world)  
Hello, world!  
NIL
```

You'll also probably want to save the file you've been working on; in the `hello.lisp` buffer, type `C-x C-s` to invoke the Emacs command `save-buffer`.

Now to try reloading this function from the source file, you'll need to quit Lisp and restart. To quit you can use a SLIME shortcut: at the REPL, type a comma. At the bottom of the Emacs window, you will be prompted for a command. Type `quit` (or `sayoonara`), and then hit

Enter. This will quit Lisp and close all the buffers created by SLIME such as the REPL buffer.¹²
Now restart SLIME by typing `M-x slime`.

Just for grins, you can try to invoke `hello-world`.

```
CL-USER> (hello-world)
```

At that point SLIME will pop up a new buffer that starts with something that looks like this:

```
attempt to call `HELLO-WORLD' which is an undefined function.  
[Condition of type UNDEFINED-FUNCTION]
```

Restarts:

```
0: [TRY-AGAIN] Try calling HELLO-WORLD again.  
1: [RETURN-VALUE] Return a value instead of calling HELLO-WORLD.  
2: [USE-VALUE] Try calling a function other than HELLO-WORLD.  
3: [STORE-VALUE] Setf the symbol-function of HELLO-WORLD and call it again.  
4: [ABORT] Abort handling SLIME request.  
5: [ABORT] Abort entirely from this process.
```

Backtrace:

```
0: (SWANK::DEBUG-IN-EMACS #<UNDEFINED-FUNCTION @ #x716b082a>)  
1: ((FLET SWANK:SWANK-DEBUGGER-HOOK SWANK::DEBUG-IT))  
2: (SWANK:SWANK-DEBUGGER-HOOK #<UNDEFINED-FUNCTION @ #x716b082a> #<Function SWANK-DEBUGGER-  
3: (ERROR #<UNDEFINED-FUNCTION @ #x716b082a>)  
4: (EVAL (HELLO-WORLD))  
5: (SWANK::EVAL-REGION "(hello-world)  
" T)
```

Blammo! What happened? Well, you tried to invoke a function that doesn't exist. But despite the burst of output, Lisp is actually handling this situation gracefully. Unlike Java or Python, Common Lisp doesn't just bail--throwing an exception and unwinding the stack. And it definitely doesn't dump core just because you tried to invoke a missing function. Instead Lisp drops you into the debugger.

While you're in the debugger you still have full access to Lisp, so you can evaluate expressions to examine the state of our program and maybe even fix things. For now don't worry about that; just type `q` to exit the debugger and get back to the REPL. The debugger buffer will go away, and the REPL will show this:

```
CL-USER> (hello-world)  
; Evaluation aborted  
CL-USER>
```

There's obviously more that can be done from within the debugger than just abort--we'll see, for instance, in Chapter 19 how the debugger integrates with the error handling system. For now, however, the important thing to know is that you can always get out of it, and back to the REPL, by typing `q`.

Back at the REPL you can try again. Things blew up because Lisp didn't know the definition of `hello-world`. So you need to let Lisp know about the definition we saved in the file `hello.lisp`. You have several ways you could do this. You could switch back to the buffer containing the file (type `C-x b` and then enter `hello.lisp` when prompted) and recompile the definition as you did before with `C-c C-c`. Or you can load the whole file, which would be a more convenient approach if the file contained a bunch of definitions, using the `LOAD` function at the REPL like this:

```
CL-USER> (load "hello.lisp")  
; Loading /home/peter/my-lisp-programs/hello.lisp  
T
```

The **T** means everything loaded correctly.¹³ Loading a file with **LOAD** is essentially equivalent to typing each of the expressions in the file at the REPL in the order they appear in the file, so after the call to **LOAD**, `hello-world` should be defined:

```
CL-USER> (hello-world)
Hello, world!
NIL
```

Another way to load a file's worth of definitions is to compile the file first with **COMPILE-FILE** and then **LOAD** the resulting compiled file, called a *FASL file*, which is short for *fast-load file*. **COMPILE-FILE** returns the name of the FASL file, so we can compile and load from the REPL like this:

```
CL-USER> (load (compile-file "hello.lisp"))
;;; Compiling file hello.lisp
;;; Writing fasl file hello.fasl
;;; Fasl write complete
; Fast loading /home/peter/my-lisp-programs/hello.fasl
T
```

SLIME also provides support for loading and compiling files without using the REPL. When you're in a source code buffer, you can use `C-c C-l` to load the file with `slime-load-file`. Emacs will prompt for the name of a file to load with the name of the current file already filled in; you can just hit Enter. Or you can type `C-c C-k` to compile and load the file represented by the current buffer. In some Common Lisp implementations, compiling code this way will make it quite a bit faster; in others, it won't, typically because they always compile everything.

This should be enough to give you a flavor of how Lisp programming works. Of course I haven't covered all the tricks and techniques yet, but you've seen the essential elements--interacting with the REPL trying things out, loading and testing new code, tweaking and debugging. Serious Lisp hackers often keep a Lisp image running for days on end, adding, redefining, and testing bits of their program incrementally.

Also, even when the Lisp app is deployed, there's often still a way to get to a REPL. You'll see in Chapter 26 how you can use the REPL and SLIME to interact with the Lisp that's running a Web server at the same time as it's serving up Web pages. It's even possible to use SLIME to connect to a Lisp running on a different machine, allowing you--for instance--to debug a remote server just like a local one.

An even more impressive instance of remote debugging occurred on NASA's 1998 Deep Space 1 mission. A half year after the space craft launched, a bit of Lisp code was going to control the spacecraft for two days while conducting a sequence of experiments. Unfortunately, a subtle race condition in the code had escaped detection during ground testing and was already in space. When the bug manifested in the wild--100 million miles away from Earth--the team was able to diagnose and fix the running code, allowing the experiments to complete.¹⁴ One of the programmers described it as follows:

Debugging a program running on a \$100M piece of hardware that is 100 million miles away is an interesting experience. Having a read-eval-print loop running on the spacecraft proved invaluable in finding and fixing the problem.

You're not quite ready to send any Lisp code into deep space, but in the next chapter you'll take a crack at writing a program a bit more interesting than "hello, world."

¹Superior Lisp Interaction Mode for Emacs

²If you've had a bad experience with Emacs previously, you should treat Lisp in a Box as an IDE that happens to use an Emacs-like editor as its text editor; there will be no need to become an Emacs guru to program Lisp. It is, however, orders of magnitude more enjoyable to program Lisp with an editor that has some basic Lisp awareness. At a minimum, you'll want an editor that can automatically match `()`s for you and knows how to automatically indent Lisp code. Because Emacs is itself largely written in a Lisp dialect, Elisp, it has quite a bit of support for editing Lisp code. Emacs is also deeply embedded into the history of Lisp and the culture of Lisp hackers: the original Emacs and its immediate predecessors, TECMACS and TMACS, were written by Lispers at the Massachusetts Institute of Technology (MIT). The editors on the Lisp Machines were versions of Emacs written entirely in Lisp. The first two Lisp Machine Emacs, following the hacker tradition of recursive acronyms, were EINE and ZWEI, which stood for EINE Is Not Emacs and ZWEI Was EINE Initially. Later ones used a descendant of ZWEI, named, more prosaically, ZMACS.

³Practically speaking, there's very little likelihood of the language standard itself being revised--while there are a small handful of warts that folks might like to clean up, the ANSI process isn't amenable to opening an existing standard for minor tweaks, and none of the warts that might be cleaned up actually cause anyone any serious difficulty. The future of Common Lisp standardization is likely to proceed via de facto standards, much like the "standardization" of Perl and Python--as different implementers experiment with application programming interfaces (APIs) and libraries for doing things not specified in the language standard, other implementers may adopt them or people will develop portability libraries to smooth over the differences between implementations for features not specified in the language standard.

⁴Steel Bank Common Lisp

⁵CMU Common Lisp

⁶SBCL forked from CMUCL in order to focus on cleaning up the internals and making it easier to maintain. But the fork has been amiable; bug fixes tend to propagate between the two projects, and there's talk that someday they will merge back together.

⁷The venerable "hello, world" predates even the classic Kernighan and Ritchie C book that played a big role in its popularization. The original "hello, world" seems to have come from Brian Kernighan's "A Tutorial Introduction to the Language B" that was part of the *Bell Laboratories Computing Science Technical Report #8: The Programming Language B* published in January 1973. (It's available online at <http://cm.bell-labs.com/cm/cs/who/dmr/bintro.html>.)

⁸These are some other expressions that also print the string "hello, world":

⁹Well, as you'll see when I discuss returning multiple values, it's technically possible to write expressions that evaluate to no value, but even such expressions are treated as returning NIL when evaluated in a context that expects a value.

¹⁰I'll discuss in Chapter 4 why the name has been converted to all uppercase.

¹¹You could also have entered the definition as two lines at the REPL, as the REPL reads whole expressions, not lines.

¹²SLIME shortcuts aren't part of Common Lisp--they're commands to SLIME.

¹³If for some reason the **LOAD** doesn't go cleanly, you'll get another error and drop back into the debugger. If this happens, the most likely reason is that Lisp can't find the file, probably because its idea of the current working directory isn't the same as where the file is located. In that case, you can quit the debugger by typing `q` and then use the SLIME shortcut `cd` to change Lisp's idea of the current directory--type a comma and then `cd` when prompted for a command and then the name of the directory where `hello.lisp` was saved.

¹⁴<http://www.flownet.com/gat/jpl-lisp.html>