32. Conclusion: What's Next?

I hope by now you're convinced that the title of this book isn't an oxymoron. However, it's quite likely there's some area of programming that's of great practical importance to you that I haven't discussed at all. For instance, I haven't said anything about how to develop graphical user interfaces (GUIs), how to connect to relational databases, how to parse XML, or how to write programs that act as clients for various network protocols. Similarly, I haven't discussed two topics that will become important when you write real applications in Common Lisp: optimizing your Lisp code and packaging your application for delivery.

I'm obviously not going to cover all these topics in depth in this final chapter. Instead, I'll give you a few pointers you can use to pursue whichever aspect of Lisp programming interests you most.

Finding Lisp Libraries

While the standard library of functions, data types, and macros that comes with Common Lisp is quite large, it provides only general-purpose programming constructs. Specialized tasks such as writing GUIs, talking to databases, and parsing XML require libraries beyond what are provided by the ANSI standardized language.

The easiest way to obtain a library to do something you need may be simply to check out your Lisp implementation. Most implementations provide at least some facilities not specified in the language standard. The commercial Common Lisp vendors tend to work especially hard at providing additional libraries for their implementation in order to justify their prices. Franz's Allegro Common Lisp, Enterprise Edition, for instance, comes with libraries for parsing XML, speaking SOAP, generating HTML, connecting to relational databases, and building graphical interfaces in various ways, among others. LispWorks, another prominent commercial Lisp, provides several similar libraries, including a well-regarded portable GUI toolkit, CAPI, which can be used to develop GUI applications that will run on any operating system LispWorks runs on.

The free and open-source Common Lisp implementations typically don't include quite so many bundled libraries, relying instead on portable free and open-source libraries. But even those implementations usually fill in some of the more important areas not addressed by the language standard such as networking and multithreading.

The only disadvantage of using implementation-specific libraries is that they tie you to the implementation that provides them. If you're delivering end-user apps or are deploying a server-based application on a server that you control, that may not matter a lot. But if you want to write code to share with other Lispers or if you simply don't want to be tied to a particular implementation, it's a little more annoying.

For portable libraries--portable either because they're written entirely in standard Common Lisp or because they contain appropriate read-time conditionalization to work on multiple implementations[1]--your best bet is to go to the Web. With the usual caveats about URLs going stale as soon as they're printed on paper, these are three of the best current starting points:

- Common-Lisp.net (`http://www.common-lisp.net/`) is a site that hosts free and open-source Common Lisp projects, providing version control, mailing lists, and Web hosting of project pages. In the first year and a half after the site went live, nearly a hundred projects were registered.
- The Common Lisp Open Code Collection (CLOCC) (`http://clocc.sourceforge.net/`) is a slightly older collection of free software libraries, which are intended to be portable between Common Lisp implementations and self-contained, not relying on any libraries not included in CLOCC itself.
- Cliki (`http://www.cliki.net/`) is a wiki devoted to free software in Common Lisp. While, like any wiki, it may change at any time, typically it has quite a few links to libraries as well to various open-source Common Lisp implementations. The eponymous software it runs on is also written in Common Lisp.

Linux users running the Debian or Gentoo distributions can also easily install an ever-growing number of Lisp libraries that have been packaged with those distributions' packing tools, `apt-get` on Debian and `emerge` on Gentoo.

I won't recommend any specific libraries here since the library situation is changing every day--after years of envying the library collections of Perl, Python, and Java, Common Lispers have, in the past couple of years, begun to take up the challenge of giving Common Lisp the set of libraries--both open source and commercial--that it deserves.

One area where there has been a lot of activity recently is on the GUI front. Unlike Java and C# but like Perl, Python, and C, there's no single way to develop GUIs in Common Lisp. Instead, it depends both on what Common Lisp implementation you're using and what operating system or systems you want to support.

The commercial Common Lisp implementations usually provide some way to build GUIs for the platforms they run on. Additionally, LispWorks provides CAPI, the previously mentioned, portable GUI API.

On the open-source side, you have a number of options. On Unix, you can write low-level X Windows GUIs using CLX, a pure-Common Lisp implementation of the X Windows protocol, roughly akin to xlib in C. Or you can use various bindings to higher-level APIs and toolkits such as GTK and Tk, much the way you might in Perl or Python.

Or, if you're looking for something completely different, you can check out Common Lisp Interface Manager (CLIM). A descendant of the Symbolics Lisp Machines GUI framework, CLIM is powerful but complex. Although many commercial Common Lisp implementations actually support it, it doesn't seem to have seen a lot of use. But in the past couple years, an open-source implementation of CLIM, McCLIM--now hosted at Common-Lisp.net--has been picking up steam lately, so we may be on the verge of a CLIM renaissance.

Interfacing with Other Languages

While many useful libraries can be written in "pure" Common Lisp using only the features specified in the language standard, and many more can be written in Lisp using nonstandard facilities provided by a given implementation, occasionally it's more straightforward to use an existing library written in another language, such as C.

The language standard doesn't specify a mechanism for Lisp code to call code written in another language or even require that implementations provide such a mechanism. But these days, almost all Common Lisp implementations support what's called a *Foreign Function Interface*, or FFI for short.[2] The basic job of an FFI is to allow you to give Lisp enough information to be able to link in the foreign code. Thus, if you're going to call a function from a C library, you need to tell Lisp about how to translate the Lisp objects passed to the function into C types and the value returned by the function back into a Lisp object. However, each implementation provides its own FFI, each with slightly varying capabilities and syntax. Some FFIs allow callbacks from C to Lisp, and others don't. The Universal Foreign Function Interface (UFFI) project provides a portability layer over the FFIs of more than a half dozen different Common Lisp implementations. It works by defining its own macros that expand into appropriate FFI code for the implementation it's running in. The UFFI takes a lowest common denominator approach, which means it can't take advantage of all the features of different implementations' FFIs, but it does provide a good way to build a simple Lisp wrapper around a basic C API.[3]

Make It Work, Make It Right, Make It Fast

As has been said many times, and variously attributed to Donald Knuth, C.A.R. Hoare, and Edsger Dijkstra, premature optimization is the root of all evil.[4] Common Lisp is an excellent language to program in if you want to heed this wisdom yet still need high performance. This may come as a surprise if you've heard the conventional wisdom that Lisp is slow. In Lisp's earliest days, when computers were programmed with punch cards, Lisp's high-level features may have doomed it to be slower than the competition, namely, assembly and FORTRAN. But that was a long time ago. In the meantime, Lisp has been used for everything from creating complex AI systems to writing operating systems, and a lot of work has gone into figuring out how to compile Lisp into efficient code. In this section I'll talk about some of the reasons why Common Lisp is an excellent language for writing high-performance code and some of the techniques for doing so.

The first reason that Lisp is an excellent language for writing high-performance code is, ironically enough, the dynamic nature of Lisp programming--the very thing that originally made it hard to bring Lisp's performance up to the levels achieved by FORTRAN compilers. The reason Common Lisp's dynamic features make it easier to write high-performance code is that the first step to writing efficient code is to find the right algorithms and data structures.

Common Lisp's dynamic features keep code flexible, which makes it easier to try different approaches. Given a finite amount of time to write a program, you're much more likely to end up with a high-performance version if you don't spend a lot of time getting into and out of dead ends. In Common Lisp, you can try an idea, see it's going nowhere, and move on without having spent a ton of time convincing the compiler your code is worthy of being run and then waiting for it to finish compiling. You can write a straightforward but inefficient version of a function--a *code sketch*--to determine whether your basic approach is sound and then replace that function

with a more complex but more efficient implementation if you determine that it is. And if the overall approach turns out to be flawed, then you haven't wasted a bunch of time tuning a function that's no longer needed, which means you have more time to find a better approach.

The next reason Common Lisp is a good language for developing high-performance software is that most Common Lisp implementations come with mature compilers that generate quite efficient machine code. I'll talk in a moment about how to help these compilers generate code that will be competitive with code generated by C compilers, but these implementations already are quite a bit faster than those of languages whose implementations are less mature and use simpler compilers or interpreters. Also, since the Lisp compiler is available at runtime, the Lisp programmer has some possibilities that would be hard to emulate in other languages--your programs can generate Lisp code at runtime that's then compiled into machine code and run. If the generated code is going to run enough times, this can be a big win. Or, even without using the compiler at runtime, closures give you another way to meld machine code with runtime data. For instance, the CL-PPCRE regular expression library, running in CMUCL, is faster than Perl's regular expression engine on some benchmarks, even though Perl's engine is written in highly tuned C. This is presumably because in Perl a regular expression is translated into what are essentially bytecodes that are then interpreted by the regex engine, while CL-PPCRE translates a regular expression into a tree of compiled closures that invoke each other via the normal function-calling machinery.[5]

However, even with the right algorithm and a high-quality compiler, you may not get the raw speed you need. Then it's time to think about profiling and tuning. The key, in Lisp as in any language, is to profile first to find the spots where your program is actually spending its time and then worry about speeding up those parts.[6]

You have a number of different ways to approach profiling. The language standard provides a few rudimentary tools for measuring how long certain forms take to execute. In particular, the **`TIME`** macro can be wrapped around any form and will return whatever values the form returns after printing a message to **`*TRACE-OUTPUT*`** about how long it took to run and how much memory it used. The exact form of the message is implementation defined.

You can use **`TIME`** for a bit of quick-and-dirty profiling to narrow your search for bottlenecks. For instance, suppose you have a function that's taking a long time to run and that calls two other functions--something like this:

```
(defun foo ()
  (bar)
  (baz))
```

If you want to see whether `bar` or `baz` is taking more time, you can change the definition of `foo` to this:

```
(defun foo ()
  (time (bar))
  (time (baz)))
```

Now you can call `foo`, and Lisp will print two reports, one for `bar` and one for `baz`. The form is implementation dependent; here's what it looks like in Allegro Common Lisp:

```
CL-USER> (foo)
; cpu time (non-gc) 60 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  60 msec user, 0 msec system
; real time  105 msec
; space allocation:
;  24,172 cons cells, 1,696 other bytes, 0 static bytes
; cpu time (non-gc) 540 msec user, 10 msec system
; cpu time (gc)     170 msec user, 0 msec system
; cpu time (total)  710 msec user, 10 msec system
; real time  1,046 msec
; space allocation:
;  270,172 cons cells, 1,696 other bytes, 0 static bytes
```

Of course, that'd be a bit easier to read if the output included a label. If you use this technique a lot, it might be worth defining your own macro like this:

```
(defmacro labeled-time (form)
  `(progn
    (format *trace-output* "~2&~a" ',form)
    (time ,form)))
```

If you replace **TIME** with `labeled-time` in `foo`, you'll get this output:

```
CL-USER> (foo)

(BAR)
; cpu time (non-gc) 60 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  60 msec user, 0 msec system
; real time  131 msec
; space allocation:
;  24,172 cons cells, 1,696 other bytes, 0 static bytes

(BAZ)
; cpu time (non-gc) 490 msec user, 0 msec system
; cpu time (gc)     190 msec user, 10 msec system
; cpu time (total)  680 msec user, 10 msec system
; real time  1,088 msec
; space allocation:
;  270,172 cons cells, 1,696 other bytes, 0 static bytes
```

From this output, it's clear that most of the time in `foo` is spent in `baz`.

Of course, the output from **TIME** gets a bit unwieldy if the form you want to profile is called repeatedly. You can build your own measurement tools using the functions **GET-INTERNAL-REAL-TIME** and **GET-INTERNAL-RUN-TIME**, which return a number that increases by the value of the constant **INTERNAL-TIME-UNITS-PER-SECOND** each second. **GET-INTERNAL-REAL-TIME** measures *wall time*, the actual amount of time elapsed, while **GET-INTERNAL-RUN-TIME** measures some implementation-defined value such as the amount of time Lisp was actually executing or the time Lisp was executing user code and not internal bookkeeping such as the garbage collector. Here's a trivial but useful profiling tool built with a few macros and **GET-INTERNAL-RUN-TIME**:

```
(defparameter *timing-data* ())

(defmacro with-timing (label &body body)
  (with-gensyms (start)
    `(let ((,start (get-internal-run-time)))
      (unwind-protect (progn ,@body)
        (push (list ',label ,start (get-internal-run-time)) *timing-data*)))))

(defun clear-timing-data ()
  (setf *timing-data* ()))

(defun show-timing-data ()
  (loop for (label time count time-per %-of-total) in (compile-timing-data) do
```

```
        (format t "~3d% ~a: ~d ticks over ~d calls for ~d per.~%"
                %-of-total label time count time-per)))

(defun compile-timing-data ()
  (loop with timing-table = (make-hash-table)
     with count-table = (make-hash-table)
     for (label start end) in *timing-data*
     for time = (- end start)
     summing time into total
     do
       (incf (gethash label timing-table 0) time)
       (incf (gethash label count-table 0))
     finally
       (return
         (sort
          (loop for label being the hash-keys in timing-table collect
               (let  ((time (gethash label timing-table))
                      (count (gethash label count-table)))
                 (list label time count (round (/ time count)) (round (* 100 (/ time total)
          #'> :key #'fifth))))
```

This profiler lets you wrap a `with-timing` around any form; each time the form is executed, the time it starts and the time it ends are recorded, associating with a label you provide. The function `show-timing-data` dumps out a table showing how much time was spent in different labeled sections of code like this:

```
CL-USER> (show-timing-data)
 84% BAR: 650 ticks over 2 calls for 325 per.
 16% FOO: 120 ticks over 5 calls for 24 per.
NIL
```

You could obviously make this profiling code more sophisticated in many ways. Alternatively, your Lisp implementation most likely provides its own profiling tools, which, since they have access to the internals of the implementation, can get at information not necessarily available to user-level code.

Once you've found the bottleneck in your code, you can start tuning. The first thing you should try, of course, is to find a more efficient basic algorithm--that's where the big gains are to be had. But assuming you're already using an appropriate algorithm, then it's down to *code bumming*--locally optimizing the code so it does absolutely no more work than necessary.

The main tools for code bumming in Common Lisp are its optional declarations. The basic idea behind declarations in Common Lisp is that they're used to give the compiler information it can use in a variety of ways to generate better code.

For a simple example, consider this Common Lisp function:

```
(defun add (x y) (+ x y))
```

I mentioned in Chapter 10 that if you compare the performance of this function Lisp to the seemingly equivalent C function:

```
int add (int x, int y) { return x + y; }
```

you'll likely find the Common Lisp version to be quite a bit slower, even if your Common Lisp implementation features a high-quality native compiler.

That's because the Common Lisp version is doing a lot more--the Common Lisp compiler doesn't even know that the values of `a` and `b` are numbers and so has to generate code to check at runtime. And once it determines they *are* numbers, it has to determine what types of numbers--

integers, rationals, floating point, or complex--and dispatch to the appropriate addition routine for the actual types. And even if `a` and `b` are integers--the case you care about--then the addition routine has to account for the possibility that the result may be too large to represent as a *fixnum*, a number that can be represented in a single machine word, and thus it may have to allocate a *bignum* object.

In C, on the other hand, because the type of all variables are declared, the compiler knows exactly what kind of values `a` and `b` will hold. And because C's arithmetic simply overflows when the result of an addition is too large to represent in whatever type is being returned, there's no checking for overflow and no allocation of a bignum object to represent the result when the mathematical sum is too large to fit in a machine word.

Thus, while the behavior of the Common Lisp code is much more likely to be mathematically correct, the C version can probably be compiled down to one or two machine instructions. But if you're willing to give the Common Lisp compiler the same information the C compiler has about the types of arguments and return values and to accept certain C-like compromises in terms of generality and error checking, the Common Lisp function can also be compiled down to an instruction or two.

That's what declarations are for. The main use of declarations is to tell the compiler about the types of variables and other expressions. For instance, you could tell the compiler that the arguments to `add` are both fixnums by writing the function like this:

```
(defun add (x y)
  (declare (fixnum x y))
  (+ x y))
```

The **`DECLARE`** expression isn't a Lisp form; rather, it's part of the syntax of the **`DEFUN`** and must appear before any other code in the function body.[7] This declaration declares that the arguments passed for the parameters `x` and `y` will always be fixnums. In other words, it's a promise to the compiler, and the compiler is allowed to generate code on the assumption that whatever you tell it is true.

To declare the type of the value returned, you can wrap the form `(+ x y)` in the **`THE`** special operator. This operator takes a type specifier, such as **`FIXNUM`**, and a form and tells the compiler the form will evaluate to the given type. Thus, to give the Common Lisp compiler all the information about `add` that the C compiler gets, you can write it like this:

```
(defun add (x y)
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

However, even this version needs one more declaration to give the Common Lisp compiler the same license as the C compiler to generate fast but dangerous code. The **`OPTIMIZE`** declaration is used to tell the compiler how to balance five qualities: the speed of the code generated; the amount of runtime error checking; the memory usage of the code, both in terms of code size and runtime memory usage; the amount of debugging information kept with the code; and the speed of the compilation process. An **`OPTIMIZE`** declaration consists of one or more lists, each containing one of the symbols **`SPEED`**, **`SAFETY`**, **`SPACE`**, **`DEBUG`**, and **`COMPILATION-SPEED`**, and a number from zero to three, inclusive. The number specifies the

relative weighting the compiler should give to the corresponding quality, with `3` being the most important and `0` meaning not important at all. Thus, to make Common Lisp compile `add` more or less like a C compiler would, you can write it like this:

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Of course, now the Lisp version suffers from many of the same liabilities as the C version--if the arguments passed aren't fixnums or if the addition overflows, the result will be mathematically incorrect or worse. Also, if someone calls `add` with a wrong number of arguments, it may not be pretty. Thus, you should use these kinds of declarations only after your program is working correctly. And you should add them only where profiling shows they'll make a difference. If you're getting reasonable performance without them, leave them out. But when profiling shows you a real hot spot in your code and you need to tune it up, go ahead. Because you can use declarations this way, it's rarely necessary to rewrite code in C just for performance reasons; FFIs are used to access existing C code, but declarations are used when C-like performance is needed. Of course, how close you can get the performance of a given piece of Common Lisp code to C and C++ depends mostly on how much like C you're willing to make it.

Another code-tuning tool built into Lisp is the function **`DISASSEMBLE`**. The exact behavior of this function is implementation dependent because it depends on how the implementation compiles code--whether to machine code, bytecodes, or some other form. But the basic idea is that it shows you the code generated by the compiler when it compiled a specific function.

Thus, you can use **`DISASSEMBLE`** to see whether your declarations are having any effect on the code generated. And if your Lisp implementation uses a native compiler and you know your platform's assembly language, you can get a pretty good sense of what's actually going on when you call one of your functions. For instance, you could use **`DISASSEMBLE`** to get a sense of the difference between the first version of `add`, with no declarations, and the final version. First, define and compile the original version.

```
(defun add (x y) (+ x y))
```

Then, at the REPL, call **`DISASSEMBLE`** with the name of the function. In Allegro, it shows the following assembly-language-like dump of the code generated by the compiler:

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y

;; code start: #x737496f4:
   0: 55         pushl  ebp
   1: 8b ec      movl   ebp,esp
   3: 56         pushl  esi
   4: 83 ec 24   subl   esp,$36
   7: 83 f9 02   cmpl   ecx,$2
  10: 74 02      jz     14
  12: cd 61      int    $97   ; SYS::TRAP-ARGERR
  14: 80 7f cb 00 cmpb  [edi-53],$0        ; SYS::C_INTERRUPT-PENDING
  18: 74 02      jz     22
  20: cd 64      int    $100  ; SYS::TRAP-SIGNAL-HIT
  22: 8b d8      movl   ebx,eax
  24: 0b da      orl    ebx,edx
  26: f6 c3 03   testb  bl,$3
  29: 75 0e      jnz    45
  31: 8b d8      movl   ebx,eax
```

```
  33: 03 da    addl     ebx,edx
  35: 70 08    jo       45
  37: 8b c3    movl     eax,ebx
  39: f8         clc
  40: c9         leave
  41: 8b 75 fc movl     esi,[ebp-4]
  44: c3         ret
  45: 8b 5f 8f movl     ebx,[edi-113]    ; EXCL::+_2OP
  48: ff 57 27 call     *[edi+39]   ; SYS::TRAMP-TWO
  51: eb f3    jmp      40
  53: 90         nop
; No value
```

Clearly, there's a bunch of stuff going on here. If you're familiar with x86 assembly language, you can probably tell what. Now compile this version of `add` with all the declarations.

```
(defun add (x y)
  (declare (optimize (speed 3) (safety 0)))
  (declare (fixnum x y))
  (the fixnum (+ x y)))
```

Now disassemble `add` again, and see if the declarations had any effect.

```
CL-USER> (disassemble 'add)
;; disassembly of #<Function ADD>
;; formals: X Y

;; code start: #x7374dc34:
   0: 03 c2    addl     eax,edx
   2: f8         clc
   3: 8b 75 fc movl     esi,[ebp-4]
   6: c3         ret
   7: 90         nop
; No value
```

Looks like they did.

Delivering Applications

Another topic of practical importance, which I didn't talk about elsewhere in the book, is how to deliver software written in Lisp. The main reason I neglected this topic is because there are many different ways to do it, and which one is best for you depends on what kind of software you need to deliver to what kind of user with what Common Lisp implementation. In this section I'll give an overview of some of the different options.

If you've written code you want to share with fellow Lisp programmers, the most straightforward way to distribute it is as source code.[8] You can distribute a simple library as a single source file, which programmers can **`LOAD`** into their Lisp image, possibly after compiling it with **`COMPILE-FILE`**.

More complex libraries or applications, broken up across multiple source files, pose an additional challenge--in order to load and compile the code, the files need to be loaded and compiled in the correct order. For instance, a file containing macro definitions must be loaded before you can compile files that use those macros. And a file containing **`DEFPACKAGE`** forms must be loaded before any files that use those packages can even be **`READ`**. Lispers call this the *system definition* problem and typically handle it with tools called *system definition facilities* or *system definition utilities*, which are somewhat analogous to build tools such as `make` or `ant`. As with `make` and `ant`, system definition tools allow you to specify the dependencies between

different files and then take care of loading and compiling the files in the correct order while trying to do only work that's necessary--recompiling only files that have changed, for example.

These days the most widely used system definition tool is ASDF, which stands for *Another System Definition Facility*.[9] The basic idea behind ASDF is that you define systems in ASD files, and ASDF provides a number of operations on systems such as loading them or compiling them. A system can also be defined to depend on other systems, which will be loaded as necessary. For instance, the following shows the contents of `html.asd`, the ASD file for the FOO library from Chapters 31 and 32:

```
(defpackage :com.gigamonkeys.html-system (:use :asdf :cl))
(in-package :com.gigamonkeys.html-system)

(defsystem html
  :name "html"
  :author "Peter Seibel <peter@gigamonkeys.com>"
  :version "0.1"
  :maintainer "Peter Seibel <peter@gigamonkeys.com>"
  :license "BSD"
  :description "HTML and CSS generation from sexps."
  :long-description ""
  :components
  ((:file "packages")
   (:file "html" :depends-on ("packages"))
   (:file "css" :depends-on ("packages" "html")))
  :depends-on (:macro-utilities))
```

If you add a symbolic link to this file from a directory listed in `asdf:*central-registry*`,[10] then you can type this:

```
(asdf:operate 'asdf:load-op :html)
```

to compile and load the files `packages.lisp`, `html.lisp`, and `html-macros.lisp` in the correct order after first making sure the `:macro-utilities` system has been compiled and loaded. For other examples of ASD files, you can look at this book's source code--the code from each practical chapter is defined as a system with appropriate intersystem dependencies expressed in the ASD files.

Most free and open-source Common Lisp libraries you'll find will come with an ASD file. Some will use other system definition tools such as the slightly older MK:DEFSYSTEM or even utilities devised by the library's author, but the tide seems to be turning in the direction of ASDF.[11]

Of course, while ASDF makes it easy for Lispers to install Lisp libraries, it's not much help if you want to package an application for an end user who doesn't know or care about Lisp. If you're delivering a pure end-user application, presumably you want to provide something the user can download, install, and run without having to know anything about Lisp. You can't expect them to separately download and install a Lisp implementation. And you want them to be able to run your application just like any other application--by double-clicking an icon on Windows or OS X or by typing the name of the program at the command line on Unix.

However, unlike C programs, which can typically rely on certain shared libraries (DLLs on Windows) that make up the C "runtime" being present as part of the operating system, Lisp programs must include a Lisp runtime, that is, the same program you run when you start Lisp though perhaps with certain functionality not needed to run the application excised.

To further complicate matters, *program* isn't really well defined in Lisp. As you've seen throughout this book, the process of developing software in Lisp is an incremental process that involves making changes to the set of definitions and data living in your Lisp image. The "program" is just a particular state of the image arrived at by loading the `.lisp` or `.fasl` files that contain code that creates the appropriate definitions and data. You could, then, distribute a Lisp application as a Lisp runtime plus a bunch of FASL files and an executable that starts the runtime, loads the FASLs, and somehow invokes the appropriate starting function. However, since actually loading the FASLs can take some time, especially if they have to do any computation to set up the state of the world, most Common Lisp implementations provide a way to *dump an image*--to save the state of a running Lisp to a file called an *image file* or sometimes a *core*. When a Lisp runtime starts, the first thing it does is load an image file, which it can do in much less time than it'd take to re-create the state by loading FASL files.

Normally the image file is a default image containing only the standard packages defined by the language and any extras provided by the implementation. But with most implementations, you have a way to specify a different image file. Thus, instead of packaging an app as a Lisp runtime plus a bunch of FASLs, you can package it as a Lisp runtime plus a single image file containing all the definitions and data that make up your application. Then all you need is a program that launches the Lisp runtime with the appropriate image file and invokes whatever function serves as the entry point to the application.

This is where things get implementation and operating-system dependent. Some Common Lisp implementations, in particular the commercial ones such as Allegro and LispWorks, provide tools for building such an executable. For instance, Allegro's Enterprise Edition provides a function `excl:generate-application` that creates a directory containing the Lisp runtime as a shared library, an image file, and an executable that starts the runtime with the given image. Similarly, the LispWorks Professional Edition "delivery" mechanism allows you to build single-file executables of your programs. On Unix, with the various free and open-source implementations, you can do essentially the same thing except it's probably easier to use a shell script to start everything.

And on OS X things are even better--since all applications on OS X are packaged as `.app` bundles, which are essentially directories with a certain structure, it's not all that difficult to package all the parts of a Lisp application as a double-clickable `.app` bundle. Mikel Evins's Bosco tool makes it easy to create `.app` bundles for applications running on OpenMCL.

Of course, another popular way to deliver applications these days is as server-side applications. This is a niche where Common Lisp can really excel--you can pick a combination of operating system and Common Lisp implementation that works well for you, and you don't have to worry about packaging the application to be installed by an end user. And Common Lisp's interactive debugging and development features make it possible to debug and upgrade a live server in ways that either just aren't possible in a less dynamic language or would require you to build a lot of specific infrastructure.

Where to Go Next

So, that's it. Welcome to the wonderful world of Lisp. The best thing you can do now--if you haven't already--is to start writing your own Lisp code. Pick a project that interests you, and do it in Common Lisp. Then do another. Lather, rinse, repeat.

However, if you need some further pointers, this section offers some places to go. For starters, check out the *Practical Common Lisp* Web site at `http://www.gigamonkeys.com/book/`, where you can find the source code from the practical chapters, errata, and links to other Lisp resources on the Web.

In addition to the sites I mentioned in the "Finding Lisp Libraries" section, you may also want explore the Common Lisp HyperSpec (a.k.a. the HyperSpec or CLHS), an HTML version of the ANSI language standard prepared by Kent Pitman and made available by LispWorks at `http://www.lispworks.com/documentation/HyperSpec/index.html`. The HyperSpec is by no means a tutorial, but it's as authoritative a guide to the language as you can get without buying a printed copy of the standard from ANSI and much more convenient for day-to-day use.[12]

If you want to get in touch with other Lispers, `comp.lang.lisp` on Usenet and the `#lisp` IRC channel or the Freenode network (`http://www.freenode.net`) are two of the main online hang- outs. There are also a number of Lisp-related blogs, most of which are aggregated on Planet Lisp at `http://planet.lisp.org/`.

And keep your eyes peeled in all those forums for announcements of local Lisp users get-togethers in your area--in the past few years, Lispnik gatherings have popped up in cities around the world, from New York to Oakland, from Cologne to Munich, and from Geneva to Helsinki.

If you want to stick to books, here are a few suggestions. For a nice thick reference book to stick on your desk, grab *The ANSI Common Lisp Reference Book* edited by David Margolies (Apress, 2005).[13]

For more on Common Lisp's object system, you can start with *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS* by Sonya E. Keene (Addison-Wesley, 1989). Then if you really want to become an object wizard or just to stretch your mind in interesting ways, read *The Art of the Metaobject Protocol* by Gregor Kiczales, Jim des Rivi&#xfffd;res, and Daniel G. Bobrow (MIT Press, 1991). This book, also known as AMOP, is both an explanation of what a metaobject protocol is and why you want one and the de facto standard for the metaobject protocol supported by many Common Lisp implementations.

Two books that cover general Common Lisp technique are *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp* by Peter Norvig (Morgan Kaufmann, 1992) and *On Lisp: Advanced Techniques for Common Lisp* by Paul Graham (Prentice Hall, 1994). The former provides a solid introduction to artificial intelligence techniques while teaching quite a bit about how to write good Common Lisp code, and the latter is especially good in its treatment of macros.

If you're the kind of person who likes to know how things work down to the bits, *Lisp in Small Pieces* by Christian Queinnec (Cambridge University Press, 1996) provides a nice blend of

programming language theory and practical Lisp implementation techniques. While it's primarily focused on Scheme rather than Common Lisp, the same principles apply.

For folks who want a little more theoretical look at things--or who just want to know what it's like to be a freshman comp sci student at M.I.T.--*Structure and Interpretation of Computer Programs*, Second Edition, by Harold Abelson, Gerald Jay Sussman, and Julie Sussman (M.I.T. *Press, 1996) is a classic computer science text that uses Scheme to teach important programming c*oncepts. Any programmer can learn a lot from this book--just remember that there are important differences between Scheme and Common Lisp.

Once you've wrapped your mind around Lisp, you may want to place it in a bit of context. Since no one can claim to really understand object orientation who doesn't know something about Smalltalk, you might want to start with *Smalltalk-80: The Language* by Adele Goldberg and David Robson (Addison Wesley, 1989), the standard introduction to the core of Smalltalk. After that, *Smalltalk Best Practice Patterns* by Kent Beck (Prentice Hall, 1997) is full of good advice aimed at Smalltalkers, much of which is applicable to any object-oriented language.

And at the other end of the spectrum, *Object-Oriented Software Construction* by Bertrand Meyer (Prentice Hall, 1997) is an excellent exposition of the static language mind-set from the inventor of Eiffel, an oft-overlooked descendant of Simula and Algol. It contains much food for thought, even for programmers working with dynamic languages such as Common Lisp. In particular, Meyer's ideas about Design By Contract can shed a lot of light on how one ought to use Common Lisp's condition system.

Though not about computers per se, *The Wisdom of Crowds: Why the Many Are Smarter Than the Few and How Collective Wisdom Shapes Business, Economies, Societies, and Nations* by James Surowiecki (Doubleday, 2004) contains an excellent answer to the question, "If Lisp's so great how come everybody isn't using it?" See the section on "Plank-Road Fever" starting on page 53.

And finally, for some fun, and to learn about the influence Lisp and Lispers have had on hacker culture, dip into (or read from cover to cover) *The New Hacker's Dictionary*, Third Edition, compiled by Eric S. Raymond (MIT Press, 1996) and based on the original *The Hacker's Dictionary* edited by Guy Steele (Harper & Row, 1983).

But don't let all these suggestions interfere with your programming--the only way to really learn a language is to use it. If you've made it this far, you're certainly ready to do that. Happy hacking!

---

[1]The combination of Common Lisp's read-time conditionalization and macros makes it quite feasible to develop portability libraries that do nothing but provide a common API layered over whatever API different implementations provide for facilities not specified in the language standard. The portable pathname library from Chapter 15 is an example of this kind of library, albeit to smooth over differences in interpretation of the standard rather than implementation-dependent APIs.

[2]A Foreign Function Interface is basically equivalent to JNI in Java, XS in Perl, or the extension module API in Python.

[3]As of this writing, the two main drawbacks of UFFI are the lack of support for callbacks from C into Lisp, which many but not all implementations' FFIs support, and the lack of support for CLISP, whose FFI is quite good but different enough from the others as to not fit easily into the UFFI model.

[4]Knuth has used the saying several times in publications, including in his 1974 ACM Turing Award paper, "Computer Programming as an Art," and in his paper "Structured Programs with goto Statements." In his paper "The Errors of TeX," he attributes the saying to C.A.R. Hoare. And Hoare, in an 2004 e-mail to Hans Genwitz of phobia.com, said he didn't remember the origin of the saying but that he might have attributed it to Dijkstra.

[5]CL-PPCRE also takes advantage of another Common Lisp feature I haven't discussed, *compiler macros*. A compiler macro is a special kind of macro that's given a chance to optimize calls to a specific function by transforming calls to that function into more efficient code. CL-PPCRE defines compiler macros for its functions that take regular expression arguments. The compiler macros optimize calls to those functions in which the regular expression is a constant value by parsing the regular expression at compile time rather than leaving it to be done at runtime. Look up **`DEFINE-COMPILER-MACRO`** in your favorite Common Lisp reference for more information about compiler macros.

[6]The word *premature* in "premature optimization" can pretty much be defined as "before profiling." Remember that even if you can speed up a piece of code to the point where it takes literally no time to run, you'll still speed up your program only by whatever percentage of time it spent in that piece of code.

[7]Declarations can appear in most forms that introduce new variables, such as **`LET`**, **`LET*`**, and the **`DO`** family of looping macros. **`LOOP`** has its own syntax for declaring the types of loop variables. The special operator **`LOCALLY`**, mentioned in Chapter 20, does nothing but create a scope in which you can make declarations.

[8]The FASL files produced by **`COMPILE-FILE`** are implementation dependent and may or may not be compatible between different versions of the same Common Lisp implementation. Thus, they're not a very good way to distribute Lisp code. The one time they can be handy is as a way of providing patches to be applied to an application running in a known version of a particular implementation. Applying the patch simply entails **`LOAD`**ing the FASL, and because a FASL can contain arbitrary code, it can be used to upgrade existing data as well as to provide new code definitions.

[9]ASDF was originally written by Daniel Barlow, one of the SBCL developers, and has been included as part of SBCL for a long time and also distributed as a stand-alone library. It has recently been adopted and included in other implementations such as OpenMCL and Allegro.

[10]On Windows, where there are no symbolic links, it works a little bit differently but roughly the same.

[11]Another tool, ASDF-INSTALL, builds on top of ASDF and MK:DEFSYSTEM, providing an easy way to automatically download and install libraries from the network. The best starting point for learning about ASDF-INSTALL is Edi Weitz's "A tutorial for ASDF-INSTALL" (`http:// www.weitz.de/asdf-install/`).

[12]SLIME incorporates an Elisp library that allows you to automatically jump to the HyperSpec entry for any name defined in the standard. You can also download a complete copy of the HyperSpec to keep locally for offline browsing.

[13]Another classic reference is *Common Lisp: The Language* by Guy Steele (Digital Press, 1984 and 1990). The first edition, a.k.a. CLtL1, was the de facto standard for the language for a number of years. While waiting for the official ANSI standard to be finished, Guy Steele--who was on the ANSI committee--decided to release a second edition to bridge the gap between CLtL1 and the eventual standard. The second edition, now known as CLtL2, is essentially a snapshot of the work of the standardization committee taken at a particular moment in time near to, but not quite at, the end of the standardization process. Consequently, CLtL2 differs from the standard in ways that make it not a very good day-to-day reference. It is, however, a useful historical document, particularly because it includes documentation of some features that were dropped from the standard before it was finished as well as commentary that isn't part of the standard about why certain features are the way they are.