CHAPTER 1

Preliminaries

Choosing a Lisp

Let's start with some practical issues: picking a Common Lisp implementation, running it, coping when things go wrong, and turning for help when coping is no longer an option. With these out the way we can then embark on the language itself in Chapter 2.

There are several competing implementations of Common Lisp available today. Dan Weinreb in his truly excellent paper *Common Lisp Implementations: A Survey* which you'll find at

http://common-lisp.net/~dlw/LispSurvey.html

lists eleven of them. Our first question is: how to tell them apart? I can't tell you which one to use; the decision is yours and depends on your requirements. Ultimately I refer you to Dan's survey and to the websites of the implementations he lists. I will though suggest criteria to help you choose, in the sections below. I'll follow this with a very brief summary of the eleven implementations.

Picking an implementation so you can learn CL is different from choosing which Lisp to use for the delivery of a major project. I'll try to guide you through both.

Implementing Common Lisp

Barring defects, which we hope are few and far between, all eleven Common Lisps listed in the survey implement and conform to *ANSI INCITS 226-1994 American National Standard for Programming Language Common LISP*. That's quite a mouthful and I'll refer to it from now on as *the ANSI standard* (or maybe just *the standard*).

At the core language level, most of what happens in one implementation is exactly the same as what happens in another. If I call the Common Lisp function `append` with a certain set of arguments in one implementation, I have every right to expect exactly the

same results as if I'd done the same thing in another one. The standard does permit some variations though. Some of these (say, the printed representations of certain objects) are pretty inconsequential; the standard simply doesn't define what happens and you aren't supposed to worry about it. In other cases the consequences of coding incorrectly around a permitted variation could be serious but the standard will give you a way to work around it.

For example (and we'll come to this in "Working with Characters" on page 0) CL stipulates a correspondence between characters and certain integers. This correspondence typically coincides with ANSI character codes but the standard doesn't say it has to. However CL provides a means of discovering precisely what this correspondence is and if you use this correctly you can write code which will port readily (often: trivially) between implementations. The upshot, not just in this specific case, is that holding implementations up to the ANSI standard will typically not help you choose between them.

Libraries

A major question to consider—indeed it's the main subject of this book—is access to libraries. Without wishing to steal my own thunder:

- Lisp is a large language and much of what might be thought of as "library" elsewhere is "core language" here.
- Common Lisp includes access to the filesystem and the clock but that's about as far as its interactions with the outside world go.
- To a greater or lesser extent, implementations bundle libraries which allow CL to be extended massively beyond this limitation.
- While some publicly available libraries can be written in portable CL, many cannot and so have "portability layers" which allow them to target a range of implementations.
- Armed Bear CL and Embedded CL compile to the JVM and C respectively and so give immediate access to Java and C libraries.

If you have a project with specific needs in mind, you're going to have to do a bit of research. This book will help.

As far as working through the book itself is concerned: the first twelve chapters involve almost no access to external libraries and so place hardly any restrictions on which Lisp you have in front of you. (The one exception is access to the *SLIME* development environment which is covered in the next section.)

Later on I'll have to get fairly implementation-specific in places. If you want to follow everything diligently then maybe you'll end up switching to different Lisps for each of parts 3 to 6. Perhaps that's not a bad thing: the overhead won't be high and the changes will give you a good feel for how the various implementations compare.

Development environments

Lisp systems are interactive. You're going to spend a lot of time, both while learning and later on when—as I hope—you adopt Lisp as a lifestyle, talking to your Lisp. The last thing you want to do is be stuck in a raw shell. Running under a "bare" Emacs would be something of an improvement on that but from bitter experience I can assure you it's not a great one.

My reason for these sweeping statements is that Lisp is highly introspective. The language strongly encourages this and specifies a range of powerful tools to boost your productivity by cashing in on it. It expects all implementations to provide at the very least something equivalent to a *listener* (allowing you to interact with all levels of the system by typing Lisp at a prompt), some level of integration with a source *editor*, an interactive *debugger*, and an object *inspector*.

I'll introduce the listener in Chapter 2 and the debugger and inspector in Chapter 4.

The debugger and inspector, and any other introspection tools which your implementation provides, will all work at a pinch in "tty mode" but you really won't be seeing them at their best. I strongly recommend that you run Lisp in an appropriate development environment from day one.

- Some implementations (especially but not limited to: Allegro, Clozure on the Mac, LispWorks) ship with integrated development environments. In Chapter 26 we'll tour one of these (LW) in detail.
- All but one of the eleven implementations listed earlier (the exception being GNU CL) support the *SLIME* development environment which runs under Emacs. SLIME is the subject of Chapter 18.

When you're choosing a Lisp you must also think about the environment in which you're going to run it (otherwise you could be stuck with that raw shell). If you have a personal preference for a particular environment that might restrict your choice of Lisp.

Last word: I know that my insisting you use an IDE means that your initial learning curve is going to be steep. I promise it will pay back.

Performance

Some implementations place greater emphasis than others on performance: some combination of a fast compiler, fast compiled code, or a compiler which tells you why your compiled code isn't fast. You'll find further details in the survey. High-end performance may or may not be a factor in your calculations but it's unlikely to be the only one. As far as learning Lisp or following this book are concerned, it's irrelevant.

Should You Pay?

I don't want to express an opinion as to whether you should go for an open source Lisp or one of the commercial offerings.

- Open source does not imply lower quality. Some of the largest commercial CL projects have be written and deployed on open source Lisps.
- As a hideous generalization, the more you pay the more bundled libraries you'll get. Commercial Lisps also do well on access to publicly available libraries (but then so do a healthy range of the others).
- All the commercial Lisps come with commercial support—vital for many projects—but so do some of the others.
- Be aware of the difference between charging for developers (on a per-seat basis, say) and charging for runtime licenses (which in particular Corman and LispWorks don't but Allegro and Scieneer do).
- If you want access to to a commercial implementation's full source code (other than for Corman which bundles it free) you'll have to negotiate for it; you might need deep pockets.
- If you aren't sure about whether one of the commercial implementations fits your needs, get in touch with the vendors and ask for an evaluation license.
- In addition to selling a range of "editions" at various prices, both Allegro and LispWorks come with free editions: *Allegro Express* and *LispWorks Personal*. These are both slightly hobbled to prevent you from using them to deploy product but are absolutely fine for learning and experimentation.

With one exception, everything in this book can be done with Lisp implementations for which you haven't had to pay anything. The exception is that parts of Chapter 28, one of the LispWorks chapters, discuss the deployment of your code as a standalone executable; for commercial reasons LW's Personal Edition doesn't support this.

Personal Bias

I've been using LispWorks on an almost daily basis for over twenty years—I was on its development team for most of the 1990s—and naturally that has left me predisposed towards it. I've also worked with Allegro and Steel Bank CL; I trained on the now defunct Interlisp-D.

For the library chapters of this book I decided on a balance between commercial and open source offerings. The grouping of chapters suggested that should I pick four of them; I went for Allegro, Clozure, Steel Bank, and LispWorks. Omission is not a sign of guilt. I don't have access to an appropriate platform for one of the eleven surveyed Lisps; I have and I use installations for all the others.

Common Lisp Implementations

Before I list the implementations, I have two final suggestions for when you're visiting websites:

- Check the quality of documentation ("quantity" may be a reasonable measure).
- Check for signs of recent activity. How long ago was the last release? Is there anyone at home?

And here, in alphabetical order, is the list of implementations:

Allegro CL (ACL) http://www.franz.com
: Commercial implementation. 32- and 64-bit native compilers on Linux, Unix, Mac, Windows. Full windowing IDE; extensive add-on libraries.

Armed Bear CL (ABCL) http://armedbear.org
: GPL. Runs under the JVM on Linux, Unix, Mac, Windows.

CMU CL (CMUCL) http://www.cons.org/cmucl
: Open license. 32-bit native compiler on Linux, Unix, Mac. Generates code competitive in speed with C; partial IDE.

Clozure CL (previously known as OpenMCL) http://www.clozure.com
: LLGPL (see Appendix B). 32- and 64-bit native compilers on Linux, Unix, Mac. Very fast compilation speed; IDE on the Mac.

Corman CL http://www.cormanlisp.com
: Commercial implementation. 32-bit native compiler on Windows. Partial IDE (costs, after first month); the Lisp itself is free of charge.

Embedded CL (ECL) http://ecls.sourceforge.net
: LGPL. Compiles Lisp into C: 32- and 64-bit on Linux, Unix, Mac, Windows.

GNU CL (GCL) http://www.gnu.org/software/gcl
: LGCP / GPL. 32-bit native compilers on Linux (also 64-bit), Unix, Mac, Windows. Small start-up time and RAM footprint.

GNU CLISP (CLISP) http://clisp.cons.org
: GPL. 32- and 64-bit native compilers on Linux, Unix, Mac, Windows.

LispWorks (LW) http://www.lispworks.com
: Commercial implementation. 32- and 64-bit native compilers on Linux, Unix, Mac, Windows. Good range of add-on libraries; extensive windowing IDE.

Scieneer CL (SCL) http://www.scieneer.com
: Commercial implementation. 32- and 64-bit native compilers on Linux, Unix. Primary focus on high-performance scientific computing.

Steel Bank CL (SBCL) http://www.sbcl.org
: BSD / public license. 32- and 64-bit native compilers on Linux, Unix, Mac, 32-bit only on Windows. One of the highest performance CL implementations.

Exercise

λλ Choose a Lisp and install it. Locate its documentation and spend five minutes familiarizing yourself with this.

The Lisp Session

You've picked a Lisp—or maybe, as some projects do, more than one—and installed it. You've decided on an environment in which to work. What next? Well, *Lisp systems are interactive* (I'll be saying this more than once because it's quite important), so one of the topics I have to cover is how you interact with them. I'll give you enough now to get you started and we'll come back to this again later: when we discuss the "standard tools" in Chapter 4, SLIME in Chapter 18, and graphical environments in Chapter 26.

Fire it up. Shortly after you'll be looking at a *prompt*, indicating that Lisp is ready for you to start typing. The exact form of the prompt differs between every implementation. In some of them it's a single character (`>` or `?` or whatever), in others it's a bit longer. For example, the LispWorks prompt starts off as

```
CL-USER 1 >
```

and the number increments every time you get a new prompt (I'll explain the `CL-USER` bit in Chapter 8).

I'm going to use SLIME for most of the examples in these early chapters. Here's what it looks like, if I start a "session", type a number at the prompt, and press `Return`:

```
; SLIME 2009-08-21
CL-USER> 42
42
CL-USER>
```

We see: an opening banner, the prompt, the number I typed, Lisp's reply (which as I'll explain in Chapter 2 will be that same number again), and a fresh prompt showing that Lisp is ready to continue business. We'll revisit this "prompt / query / reply / prompt" cycle in Chapter 2.

The Corman IDE is a little different: there's no visible prompt and instead of `Return` you should use the `Numeric Enter` key to dispatch what you typed (on a laptop the more accessible `Shift Return` combination is set up to do the same thing).

Exercise

λ Start up a lisp session and interact with it, minimally.

As your Lisp session proceeds and you interact with the system, you are modifying its *state*. As we'll see later, even typing `42` at the prompt does that. More interesting though are the functions you define, the variables you set, the source files you compile and load, and so on. The definitions you make "augment the environment": they're with you for as long as the session lasts. In Chapter 28 I'll talk about saving a session's state back to disk (this is called *saving an image*) and about directing the image to perform some action other than presenting you with a prompt when it's restarted. If you're building any sort of application this information will be useful to you; for now all you need to know is that it's possible. Alternatively, you can close the session down and when you start another one you'll be working with a clean slate.

Ending the Session

Seeing as I just mentioned exits I might as well move on to what you have to do to perform one. Other than killing its process from the outside, that is. There are benefits to a clean shutdown: you're giving the system an opportunity to release any resources (for example, open files) before it performs the exit, and for bonus points you're learning early on how to implement your future application's "Exit" menu.

The details of this and the next few sections cover areas which are not defined by the standard. (Really: there's no standard way to exit Lisp.) While much of this book focuses on standardized features, or on how libraries work to smooth over differences between the implementations, there are times when you just have to know "how your Lisp does it". This is one of them.

- In all but two of the implementations, the "programmatic" incantation to close down the current session is `(quit)`:

```
CL-USER> (quit)
```

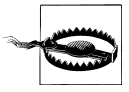
- In Allegro either `(exit)` or `:exit` will do:

```
CL-USER(1): :exit
; Exiting
[ndl@vanity ~]$
```

- In Corman it's the somewhat lengthier `(lisp-shutdown "")`.
- In the windowing IDEs: close the main window. This gesture will prompt to confirm the exit and then terminate Lisp cleanly. In LispWorks the keybinding `control-x control-c` has the same effect.

Exercise

λ Quit your lisp session.



Lisp veterans may well be wondering about "packages" here: it turns out that `(quit)` and friends are exported from packages "used by `CL-USER`" in all eleven Lisps. For novices: don't worry about this detail. Its meaning will become clear after Chapter 8.

Getting into Difficulties

I've always found that getting into difficulties is really quite straightforward. Certainly three interesting ways of doing it spring rapidly to mind. Bailing out of them is totally implementation-specific and I intend here to breeze through all the possibilities.

Unless you made a real mess earlier of typing the number `42` this will be your first brush with the Lisp *debugger* and initially your only aim will be to get out of it, to "clear the error". When you start writing proper code you'll find that the debugger is one of the most powerful tools in your Lisp environment and that "immediately getting out of it" isn't always a useful policy. We'll come back to this later and cover the debugger properly in Chapter 4 and Chapter 18.

Dealing with Errors

Under what we might call "exceptional", although not necessarily uncommon, circumstances Lisp will *signal an error*. (Quite what this phrase means I'm going to postpone for Chapter 12.) Performing arithmetic on something that's not a number, division by zero, exceeding array bounds, write access on a read-only file, anything that you the programmer have deemed to be an error,... The list is a long one, and the immediate significance of errors is that unless directed not to, Lisp responds to them by *entering the debugger*. It will print out a description of the error, offer you a number of actions to take (*restarts*), possibly tell you how to get further help, and modify the prompt to remind you that you're in trouble. Here's a simple example:

```
? rubbish
> Error: Unbound variable: RUBBISH
> While executing: CCL::TOPLEVEL-EVAL, in process listener(1).
> Type :GO to continue, :POP to abort, :R for a list of available restarts.
> If continued: Retry getting the value of RUBBISH.
> Type :? for other options.
1 >
```

The modified prompt "`1 >`" tells you that you're "one level deep" in the debugger. If another error is signaled the `1` will become a `2`, and so on. Get out of the debugger altogether and the prompt reverts (to "`?`" in this case, which incidentally is Clozure CL).

You might be wondering what the fuss is about, why you can't keep working at this modified prompt. In truth you can keep on going but you may get confusing results: any transient state which was in effect when the error occurred (examples: temporary values for global variables, files opened for output) will persist until you *unwind* that state by quitting the debugger.

Here's how you do it.

CLISP, Clozure, CMUCL, LispWorks, SBCL
: Type `:a` (a colon followed immediately by the letter `a` for "abort", either upper or lower case) and then `Return`. Example (SBCL):

```
1 > :a

?
```

Embedded CL and Scieneer CL
: Use `:q` (for "quit").

Allegro CL
: The command this time is `:pop`.

Armed Bear CL, Corman CL and GNU CL
: These Lisps use *numbered restarts*. You want the restart that offers you the *top level*. In the following example (ABCL), only one restart is offered and you invoke it by typing in its number. Corman and GCL are a little different but similar enough to this one that you shouldn't have any difficulties adapting it.

```
CL-USER(1): rubbish
Debugger invoked on condition of type UNBOUND-VARIABLE:
  The variable RUBBISH is unbound.
Restarts:
  0: TOP-LEVEL Return to top level.
[1] CL-USER(2): 0
CL-USER(3):
```

In the LispWorks IDE
: The keybinding `meta-A` (typically: the combination `alt-shift-a`) is equivalent to `:a`. These three keys are close together and you don't have to exercise your fingers much to practice this one.

SLIME
: The debugger opens in a new Emacs buffer. Type `q` to dismiss it. This straightforward keybinding is the same for every implementation and your decision to use SLIME has already started to pay for itself.

Exercise

λ Provoke an error and then get back to the top level.

Lisp is Unresponsive

The other two ways of getting into trouble have the same apparent symptom—you can't get back to the prompt—but very different causes:

- Your syntax is imbalanced in such a way that that Lisp doesn't think you've finished typing yet. Perhaps you meant to enter a literal string but never entered the closing double quote, as in the second line of this broken Lisp function:

```
(defun say-hello (person)
  (format nil "Hello, ~a
          person))
```

  Your Lisp sees the opening " and starts parsing a string. Lisp strings can be multi-lined, and so when you get to the end of what you were planning to type Lisp is still looking for the end of your string. Impasse.

  There are a few other syntax forms in Lisp which require "balancing" like this, and we'll cover them all by and by. In any case, your best bet may to abandon what you were typing before and get back to a fresh prompt.

- Lisp is stuck in some operation (maybe it's waiting for an external resource which isn't co-operating) or maybe it's just caught in a loop. You want to interrupt it and, again, get back to the prompt.

How you respond to these situations depends not so much on your Lisp but on the environment under which you're running it.

Raw shell
: If you're in a raw shell then your options are limited. Figure out what the syntax error was and start balancing out strings, lists, comments, etc. until you've got something which might be nonsense Lisp but at least looks like syntactically complete nonsense. Expect a visit to the debugger as your reward. For interrupts perhaps the shell interrupt (`control-c`) will work; if it doesn't then take a moment to consider not working in a raw shell next time, before you kill your Lisp from the outside.

SLIME
: Get back to the prompt with `control-c control-u`, send an interrupt to Lisp with `control-c control-c`. These two combinations will work for all Lisp implementations supported by SLIME.

Allegro IDE
: Recover the prompt with `f9`, interrupt Lisp with the `Break` key.

LispWorks IDE
: Get back to the prompt with `meta-k`, and use `control-Break` to send an interrupt.

In all of the above a successful interrupt will throw you into the debugger. Proceed as before to exit that and get back to the top level.

Exercise

λ Enter one precisely double quote and press Return. Now get back to the prompt.

Exercise

λ Type

```
(loop)
```

and press `Return`. Now get back to the prompt.

Looking Elsewhere

I don't pretend for an instant that a book of this nature can be self-contained. You will see me say, again and again, "go look this up". This section lists some places to visit for more information about Common Lisp as a language and for assistance when you run aground. All this is additional to the documentation which accompanies your Lisp implementation and to the resources specific to the libraries of later chapters which I'll note as we get to them.

The 1994 ANSI standard for Common Lisp is available online (the *HyperSpec*), from:

http://www.lispworks.com/documentation/HyperSpec/Front

The standard has been mirrored extensively and you may well find that your installation includes a copy or something equivalent.

Exercise

λλ Decide which online copy of the standard you're going to use. Spend five minutes floating through it to get an idea of its overall structure.

Very few people report success with using the standard as a tutorial. Its legalese is aimed at experts who already understand the basic principles. You will find it increasingly useful as a reference but it's not a starting point.

Guy L. Steele's *Common Lisp, the Language* (Digital Press) explains in a way the standard does not and is very thorough. Again though this might not be immediately accessible to beginners. Note that this book has two editions. The first, *CLtL1* (1984), is a snapshot of the evolution of CL; quite a bit changed after this was printed and you should use it only with caution. The second edition, *CLtL2* (1990), is very close to the standard and for most purposes will be good enough. It's available online here:

http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/html/cltl/cltl2.html

A number of books are aimed at Common Lisp for beginners. They devote more space than I can to the language but more or less nothing to its libraries (which is where this volume comes in). Consider Paul Graham's *ANSI Common Lisp* (Prentice Hall, 1996) —but not his earlier *On Lisp* which isn't a beginners' book at all—and Peter Seibel's *Practical Common Lisp* (Apress, 2005). Graham's book lists the differences between CLtL1/2 and the standard. Seibel's is available online from:

http://www.gigamonkeys.com/book

Help from Humans

When I was first learning Lisp I was commanded to seek help if I had made absolutely no progress with a problem for over 20 minutes. Years later I still find that number scary but there's wisdom in it.

If your difficulties concern Common Lisp I recommend the `#lisp` channel at Freenode (*irc://irc.freenode.net/*). Time zones being what they are you can probably get a response any time of the day. Some of the chat is quite highbrow but naive interruptions from novices are welcomed and treated generously.

Each implementation is minded by some combination of email lists from various perspectives: users, developers, or product support. Use whatever seems most appropriate. Don't suffer in silence.

Exercise

λ Locate your implementation's support network.