

Understanding Common Lisp

Dave Cooper

First Edition, 2026

Copyright © 2000, 2003, 2011, 2026 Franz Inc. and [Dave Cooper](#).
All rights reserved.

First edition, 2026. Derived from the third edition of
Basic Lisp Techniques, published by Franz Inc. in three
editions, in 2000, 2003 and 2011.

The source is at <https://github.com/genworks/understanding-common-lisp>.

Preface

This book began as *Basic Lisp Techniques*, which Franz Inc. published in 2000, and again in a second edition in 2003 and a third in 2011. It has been a free download from their website all that time, and it has served many programmers as a quick start in the language. The title it was meant to have was *Understanding Common Lisp*. This book, under that title, is derived from the third edition of *Basic Lisp Techniques*, and is the first edition of *Understanding Common Lisp*.

The third edition was made the hard way. By 2011 I had lost the LaTeX source of the book, and Steve Sears of Franz made that edition's changes by editing the PDF itself. For this book the source was reconstructed from that edition, and it is now kept in a public repository, at the address on the copyright page.

The language has hardly changed since 2000. Common Lisp is defined by a standard that was finished in 1994, and a program written to that standard then still runs now. What has changed is everything around the language. There are several free implementations that compile to fast native code. A library manager puts a couple of thousand open-source libraries one command away. The editor support that used to come with one vendor's product works with every implementation. And many of us now write Lisp alongside an AI agent that can evaluate code in the same running image we are working in.

Basic Lisp Techniques was written around one implementation, Allegro CL. This book is written so that the examples run on any Common Lisp. They are shown as SBCL prints them, since SBCL is free and is what most newcomers install first, and every one of them was run before it went into the book.

The book is still short. It is for a working programmer who wants to read and write Common Lisp with some confidence in a few sittings, and who will go on to the longer books and the reference material after that. *Basic Lisp Techniques* ended with a sample application. That chapter is gone: there are plenty of real programs to read online now, and Chapter 5 says where to find them.

My thanks to Franz Inc., who asked for this book in the first place and have kept it available for more than twenty-five years, and to Steve Sears, for its third edition.

Dave Cooper
<https://davecooper.name>

Foreword by Franz Inc.¹

Computers, and the software applications that power them, permeate every facet of our daily lives. From groceries to airline reservations to dental appointments, our reliance on technology is all-encompassing. And, it's not enough. Every day, our expectations of technology and software increase:

- smart appliances that can be controlled via the internet
- better search engines that generate information we actually want
- voice-activated laptops
- cars that know exactly where to go

The list is endless. Unfortunately, there is not an endless supply of programmers and developers to satisfy our insatiable appetites for new features and gadgets. Every day, hundreds of magazine and on-line articles focus on the time and people resources needed to support future technological expectations. Further, the days of unlimited funding are over. Investors want to see results, fast.

Common Lisp (CL) is one of the few languages and development options that can meet these challenges. Powerful, flexible, changeable on the fly — increasingly, CL is playing a leading role in areas with complex problem-solving demands. Engineers in the fields of bioinformatics, scheduling, data mining, document management, B2B, and E-commerce have all turned to CL to complete their applications on time and within budget. CL, however, no longer just appropriate for the most complex problems. Applications of modest complexity, but with demanding needs for fast development cycles and customization, are also ideal candidates for CL.

Other languages have tried to mimic CL, with limited success. Perl, Python, Java, C++, C# — they all incorporate some of the features that give Lisp its power, but their implementations tend to be brittle.

The purpose of this book is to showcase the features that make CL so much better than these imitators, and to give you a “quick-start” guide for using Common Lisp as a development environment. If you are an experienced programmer in languages other than Lisp, this guide gives you all the tools you need to begin writing Lisp applications. If you’ve had some exposure to Lisp in the past, this guide will help refresh those memories and shed some new light on CL for you.

¹This foreword was written by Franz Inc. for *Basic Lisp Techniques*, and is reprinted as it appeared there.

But be careful, Lisp can be addicting! This is why many Fortune 500 companies will use nothing else on their 24/7, cutting-edge, mission-critical applications. After reading this book, trying our software, and experiencing a 3 to 10 times increase in productivity, we believe you will feel the same way.

Contents

Preface	iii
Foreword by Franz Inc.	v
1 Introduction	1
1.1 The Past, Present, and Future of Common Lisp	1
1.1.1 Lisp Yesterday	1
1.1.2 Lisp Today	1
1.1.3 Lisp Tomorrow	2
1.2 Is Lisp Slow? Is It Big?	2
1.3 The CL Model of Computation	3
2 Operating a CL Development Environment	5
2.1 Choosing an Implementation	5
2.2 Installing a CL Environment	6
2.3 Running CL in a Terminal	6
2.3.1 Starting and Stopping	6
2.3.2 Compiling and Loading a File	7
2.4 Running CL inside an Editor	7
2.4.1 Emacs with SLIME or Sly	8
2.4.2 Other Editors	9
2.5 The Init File	9
2.6 Libraries: Quicklisp and ASDF	9
2.6.1 Loading Libraries with Quicklisp	9
2.6.2 Defining Your Own System with ASDF	10
2.7 Scripts and Standalone Programs	11
2.7.1 Running a File as a Script	11
2.7.2 Building an Executable	11
2.8 Debugging in CL	11
2.8.1 The Debugger	12
2.8.2 Restarts	12
2.8.3 Stopping a Program Yourself	13
2.8.4 Trace, Describe and Friends	13
2.8.5 Timing	14
2.9 Working with an AI Agent	14

3	The CL Language	17
3.1	Overview of CL and its Syntax	17
3.1.1	How Expressions Are Evaluated	17
3.1.2	Lisp Syntax Simplicity	18
3.1.3	Turning Off Evaluation	19
3.1.4	Fundamental CL Data Types	19
3.1.5	Functions	21
3.1.6	Global and Local Variables	21
3.2	The List as a Data Structure	23
3.2.1	Accessing the Elements of a List	23
3.2.2	The “Rest” of the Story	24
3.2.3	The Empty List	24
3.2.4	Are You a List?	24
3.2.5	The Conditional If	25
3.2.6	Length of a List	25
3.2.7	Member of a List	25
3.2.8	Getting Part of a List	26
3.2.9	Appending Lists	26
3.2.10	Adding Elements to a List	26
3.2.11	Removing Elements from a List	27
3.2.12	Sorting Lists	27
3.2.13	Treating a List as a Set	27
3.2.14	Mapping a Function to a List	28
3.2.15	Property Lists	28
3.3	Control of Execution	28
3.3.1	If, When and Unless	29
3.3.2	Logical Operators	29
3.3.3	Cond	29
3.3.4	Case	30
3.3.5	Iteration	30
3.4	Functions as Objects	31
3.4.1	Named Functions	31
3.4.2	Functional Arguments	31
3.4.3	Anonymous Functions	31
3.4.4	Optional Arguments	32
3.4.5	Keyword Arguments	32
3.4.6	Multiple Values	33
3.5	Input, Output, Streams, and Strings	33
3.5.1	Format	33
3.5.2	Print and Princ	34
3.5.3	Read	35
3.5.4	File Input and Output	35
3.5.5	Pathnames	35
3.6	Hash Tables, Arrays, Structures, and Classes	36
3.6.1	Hash Tables	36

3.6.2	Arrays	37
3.6.3	Structures	37
3.6.4	Classes and Methods	38
3.7	Macros	39
3.8	Conditions and Error Handling	40
3.9	Packages	41
3.9.1	The Keyword Package	42
3.10	Common Stumbling Blocks	43
3.10.1	Quotes	43
3.10.2	Function Argument Lists	43
3.10.3	Symbols vs. Strings	43
3.10.4	Equality	44
3.10.5	Distinguishing Macros from Functions	45
3.10.6	Modifying What You Should Not	45
4	Interfaces	47
4.1	The Operating System	47
4.2	Foreign Functions	48
4.3	Threads	48
4.4	TCP/IP Sockets	49
4.5	JSON	50
4.6	Databases	50
4.7	Web Serving	51
4.8	Embedded Languages	52
5	Where to Go Next	53
5.1	Reference	53
5.2	Books	53
5.3	Libraries and Examples	54
5.4	Community	54
5.5	A Last Word	54

Chapter 1

Introduction

1.1 The Past, Present, and Future of Common Lisp

1.1.1 Lisp Yesterday

John McCarthy worked out the basic principles of Lisp in 1958, at MIT. The name stood for “LISt Processing”: the language was built for manipulating symbols and lists of symbols, where the other languages of the day were built for arithmetic. A list, in this sense, is just a sequence of items, like a shopping list or a checklist.

The idea that set Lisp apart was that a Lisp program is itself written as lists. The language can therefore read, build and transform programs with the same operations it uses on any other data. Everything that makes Lisp unusual follows from that.

Lisp spread quickly through the research world and split into many dialects. In the early 1980s the people behind the main ones agreed to merge them into one language, Common Lisp. Guy Steele described it in the book *Common Lisp: the Language* in 1984, and after ten more years of work it became an ANSI standard in 1994. It was the first object-oriented language to be standardized by ANSI.

1.1.2 Lisp Today

The standard has not changed since 1994, and this turns out to be one of the best things about the language. A Common Lisp program written thirty years ago still runs today, usually without touching a line of it, on implementations and machines that did not exist when it was written. Libraries do not rot when a new version of the language comes out, because one does not come out. Code you write this year will not need rewriting for the language’s sake.

The language can stand still because it can be extended from the inside. Most of what other languages add in new versions (a new kind of loop, pattern matching, a new way to manage resources) can be added to Common Lisp by a library, and looks no different from what is built in. Section 3.7 shows how.

Meanwhile the implementations and tools have kept moving. There are several maintained implementations of Common Lisp, most of them free and open source, and the best of them compile to efficient machine code. Some two thousand open-source libraries can

be installed with one command. Common Lisp is in daily commercial use in engineering design, scheduling and logistics, finance, and research computing, and it has a small, steady community of people who chose it on purpose.

1.1.3 Lisp Tomorrow

For most of the history of software, people have hoped for tools that would let them describe a program at a high level and have the detailed code written for them. Lisp's answer has always been that the high-level description and the program should be the same thing, written in one language. A Common Lisp program can generate other Common Lisp code, normally at compile time, so you can build the language up toward your problem until the program reads like a description of it.

That matters more now, not less. A growing share of code is drafted by AI models and then checked by people. A language with a small, regular syntax, a precise standard, and a running image that can be asked what is true right now is a good place to do that kind of work: the model can try an expression and see the answer before it proposes anything, and so can you. Section 2.9 says more.

Several long-standing features of Common Lisp also save time and people:

- **Automatic memory management.** You do not allocate and free memory by hand. Lisp was the first language to have a garbage collector.
- **Dynamic typing.** Values have types, but variables do not have to. You can write a function without declaring the type of anything, and add declarations later where they help the compiler or the reader.
- **Redefinition while running.** You can add and replace functions, classes and data in a running program without stopping it. This is how Lisp is normally developed, and it also means that a server can be inspected and patched while it stays up.
- **A large standard language.** Hash tables, arbitrary-precision integers, exact fractions, multi-dimensional arrays, a full object system, formatted output, and a condition system for handling errors are all part of the standard.
- **Portability.** Because the standard is precise, code moves between implementations and operating systems with little or no change.

1.2 Is Lisp Slow? Is It Big?

The old complaints about Lisp were that it was too slow and needed too much memory. They were fair on the machines of the 1970s. They have not been true for a long time. Compiled Common Lisp runs at the speed of other compiled languages, and a Lisp image of forty megabytes is small next to a web browser tab.

The cost that matters today is the time of the people writing the program. Common Lisp teams tend to be small because one person with a good grasp of the language can build and maintain a great deal.

1.3 The CL Model of Computation

In most languages you work in a cycle: edit the source, build, run the program from the beginning, and see what happens. In Common Lisp you start the Lisp once and leave it running. You type an expression and it is evaluated at once. You change a function and send just that function to the running Lisp, which replaces the old definition and carries on, with all its data still in memory. You test the change in the next second.

This running Lisp is called the *image*, and the prompt where you talk to it is called the *REPL*, for read-eval-print loop: it reads an expression, evaluates it, prints the result, and waits for the next one. Working this way changes how you write programs. You build them in small pieces and try each piece as you go, so you find out early when you have misunderstood something.

The next chapter shows how to set this up.

Chapter 2

Operating a CL Development Environment

This chapter covers the hands-on side of working with Common Lisp: installing it, talking to it, connecting it to your editor, loading libraries, and finding out what went wrong when something does. If Lisp code is completely new to you, you may want to glance ahead at Chapter 3 as you read.

It helps to think of a running Common Lisp as a small operating system of its own, sitting on top of whichever operating system your machine runs. You start it, and then you load things into it, ask it questions, and change it while it runs. People once built whole workstations this way, with Lisp all the way down.

Have a Lisp running while you read, and type the examples in.

2.1 Choosing an Implementation

Common Lisp is a standard with several implementations, in the way that C and SQL are. The ones you are most likely to meet:

- **SBCL** (Steel Bank Common Lisp), <https://www.sbcl.org>. Free and open source, with a compiler that produces fast native code and gives thorough warnings. The most widely used implementation.
- **CCL** (Clozure CL), <https://ccl.clozure.com>. Free and open source. Compiles very quickly, and is pleasant to develop in.
- **ECL** (Embeddable Common Lisp), <https://ecl.common-lisp.dev>. Free. Compiles by way of C, so it can be linked into C and C++ programs and runs almost anywhere.
- **ABCL** (Armed Bear Common Lisp), <https://abcl.org>. Free. Runs on the Java virtual machine and can call Java libraries.
- **Allegro CL**, from Franz Inc., <https://franz.com>, and **LispWorks**, <https://www.lispworks.com>. Commercial products with their own development environments, graphical toolkits and support, each with a free edition for learning.

They all implement the same language, and what you learn on one carries over to the others. This book uses SBCL. Where something in the text is specific to SBCL, it says so.

2.2 Installing a CL Environment

SBCL is packaged for the common systems:

```
sudo apt install sbcl      (Debian, Ubuntu)
brew install sbcl          (macOS with Homebrew)
```

For Windows there is an installer at <https://www.sbcl.org>. The packaged versions can be a few releases behind; the website has current binaries if you want them.

If you use Docker, you can try an implementation without installing anything. The `clfoundation` images on Docker Hub cover most of the free implementations:

```
docker run -it -rm clfoundation/sbcl
```

2.3 Running CL in a Terminal

The simplest way to run CL is from a terminal. Few people work this way all day, because an editor connected to Lisp is far more convenient, but it is the place to start, and it is still useful for logging in to a Lisp that is running on a remote machine.

2.3.1 Starting and Stopping

To start SBCL, type

```
sbcl
```

You will see a few lines of introduction and then a prompt:

```
This is SBCL 2.6.9, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.
...
*
```

CL has entered its read-eval-print loop and is waiting for you to type something. It will read what you type, evaluate it to get a return value, and print that value:

```
* (+ 2 2)
4
```

To leave SBCL, type

```
(exit)
```

Each implementation has its own name for this (in CCL it is `(quit)`). Typing the end-of-file character, Control-D, works in most of them.

The bare terminal prompt has no command history and no line editing. If you plan to spend any time there, install `rlwrap` and start Lisp as `rlwrap sbcl`.

2.3.2 Compiling and Loading a File

Try it: With your text editor, create a file `/tmp/hello.lisp` containing the following (do not worry about understanding it yet):

```
(defun hello ()
  (write-string "Hello, World!"))
```

Now, at the CL prompt, compile and load the file. The text after each `*` prompt is what you type; everything else is printed by CL:

```
* (compile-file "/tmp/hello.lisp")
; compiling file "/tmp/hello.lisp" (written 05 OCT 2026 06:48:14 PM):

; wrote /tmp/hello.fasl
; compilation finished in 0:00:00.000
#P"/tmp/hello.fasl"
NIL
NIL
* (load "/tmp/hello")
T
```

`Compile-file` translates the source file into a binary file of machine code. The binary is called a *fasl* file, for “fast loading.” `Load` brings a file’s definitions into the running Lisp. Given a name with no extension, it picks the *fasl* file if there is one.

You can also `load` a source file directly, and the effect is the same, except that it has to be processed each time. *Fasl* files are specific to the implementation, its version, and the kind of machine. You do not pass them around; each Lisp makes its own from the source.

Now call the function you defined:

```
* (hello)
Hello, World!
"Hello, World!"
```

The text appears twice. The first line is the function printing the string, as you asked it to. The second is the REPL printing the value that the function returned, which happens to be the same string; it is shown in double quotes because the REPL prints values in the form you would type them. Chapter 3 has more on this.

The point for now is the way of working: definitions live in files, you compile and load them into a running Lisp, and you call them from the prompt.

2.4 Running CL inside an Editor

The way most people develop in CL is with the Lisp running as a subprocess of their text editor. The editor then knows about the running Lisp. With a keystroke you can send it the function you are editing, ask it for a function’s arguments or documentation, jump to the source of anything that is defined, and look inside live data.

2.4.1 Emacs with SLIME or Sly

The traditional editor for Lisp is GNU Emacs, which is itself largely written in a dialect of Lisp, and the standard connection between Emacs and Common Lisp is SLIME (<https://slime.common-lisp.dev>). Sly is a fork of SLIME with some added conveniences. Both work with every implementation listed above, and what follows applies to both.

If you are new to Emacs, run its built-in tutorial first (it is under the Help menu). Emacs commands are written as key chords: **C-x** means hold Control and press x; **M-x** means hold Meta, which is the Alt or Option key, and press x.

To install SLIME, add the MELPA package archive to Emacs if you have not already (<https://melpa.org> has the two lines to copy), then type **M-x package-install**, press Return, and give the name `slime`. Then put this line in your Emacs init file to say which Lisp to run:

```
(setq inferior-lisp-program "sbcl")
```

Now **M-x slime** starts Lisp and opens a REPL in an Emacs window. The prompt shows the current package, a notion explained in Section 3.9:

```
CL-USER> (+ 2 2)
4
```

The rest of this book shows examples with this prompt.

Table 2.1 lists the commands you will use most.

Key	Action
C-c C-c	Compile the definition the cursor is in
C-c C-k	Compile and load the whole file
C-x C-e	Evaluate the expression before the cursor
C-c C-z	Switch to the REPL
M-.	Go to the definition of the name at the cursor
M-,	Come back from there
C-c C-d d	Describe the name at the cursor
C-c RET	Expand the macro call at the cursor
M-p, M-n	In the REPL: previous and next input
C-c C-c	In the REPL: interrupt a running computation

Table 2.1: Common SLIME and Sly commands

While you type a call to a function, its argument list is shown at the bottom of the window. Emacs indents Lisp code for you and shows which parentheses match, so you read the structure of the code from its indentation and never count parentheses.

Try it: Open `/tmp/hello.lisp` in Emacs, with SLIME running. Put the cursor inside the definition of `hello` and press **C-c C-c**. Switch to the REPL with **C-c C-z** and type `(hello)`. Now go back to the file, change the string to `"Hello, CL World!"`, press **C-c C-c** again, and call `(hello)` again in the REPL. You have changed a running program without restarting it.

2.4.2 Other Editors

Emacs is not required. Visual Studio Code has the Alive extension, which gives a REPL, evaluation from the editor, a debugger and an inspector. Vim and Neovim have Vlime and Slimv. Lem (<https://lem-project.github.io>) is an Emacs-like editor written in Common Lisp, which works with Lisp as soon as it starts. Allegro CL and LispWorks each come with their own environment.

Whatever you choose, make sure it gives you the essentials: a REPL, a key to send the current definition to the running Lisp, and automatic indentation.

2.5 The Init File

When CL starts, it looks for an initialization file in your home directory and loads it. In SBCL this is `.sbclrc`; in CCL it is `.ccl-init.lisp`. The file holds ordinary Lisp, and is the place to load the tools you want in every session. The next section puts something in it.

2.6 Libraries: Quicklisp and ASDF

2.6.1 Loading Libraries with Quicklisp

Quicklisp (<https://www.quicklisp.org>) downloads and loads open-source libraries, along with whatever they depend on. To install it, download one file and load it:

```
curl -O https://beta.quicklisp.org/quicklisp.lisp
sbcl -load quicklisp.lisp
```

and then, at the Lisp prompt:

```
* (quicklisp-quickstart:install)
* (ql:add-to-init-file)
```

The second form adds a few lines to your init file so that Quicklisp is there whenever you start Lisp. From then on, one call fetches a library and loads it:

```
CL-USER> (ql:quickload "alexandria")
To load "alexandria":
  Load 1 ASDF system:
    alexandria
; Loading "alexandria"

("alexandria")
CL-USER> (alexandria:flatten '(1 (2 (3 4)) 5))
(1 2 3 4 5)
```

Alexandria, loaded here, is a collection of small utilities that most other libraries use. `(ql:system-apropos "json")` searches the library list by name. Chapter 4 uses several libraries, and Chapter 5 says where to look for more.

Quicklisp is the common starting point, and there are other tools for the same job, such as `ocicl` and `Qlot`, which add things like pinning the exact versions a project uses.

2.6.2 Defining Your Own System with ASDF

A program of any size is spread over several files, some of which have to be loaded before others. ASDF is the standard tool for describing that. It comes with every implementation mentioned in this chapter, and it is what Quicklisp uses underneath. In ASDF, a library or program is called a *system*, and is described by a file ending in `.asd`.

Here is a complete small system, in a directory named `greeter`. First `greeter.asd`:

```
(defsystem "greeter"
  :description "A very small example system."
  :version "0.1.0"
  :depends-on ("alexandria")
  :components ((:file "package")
               (:file "greet" :depends-on ("package"))))
```

then `package.lisp`:

```
(defpackage :greeter
  (:use :common-lisp)
  (:export #:greet))
```

and `greet.lisp`:

```
(in-package :greeter)

(defun greet (name)
  (format nil "Hello, ~a!" (string-capitalize name)))
```

Put the `greeter` directory inside `~/common-lisp/`, which is where ASDF looks by default, and you can load the system by name like any other library:

```
CL-USER> (ql:quickload "greeter")
To load "greeter":
  Load 1 ASDF system:
    greeter
; Loading "greeter"

("greeter")
CL-USER> (greeter:greet "reader")
"Hello, Reader!"
```

ASDF compiles each file the first time and whenever it has changed, keeps the fasl files out of your source directory, and loads the files in an order that respects the dependencies you declared. Quicklisp fetches Alexandria if it is not yet installed.

Start every project this way, however small. It costs three short files, and it means anyone, including you a year from now, can load the program with one line.

2.7 Scripts and Standalone Programs

2.7.1 Running a File as a Script

CL is normally used interactively, but it will also run a file from the command line and exit. Here is `script.lisp`, which copies its input to its output with leading spaces removed from each line:

```
;;; Print each line of standard input with leading spaces removed.
(loop for line = (read-line *standard-input* nil)
      while line
      do (write-line (string-left-trim " " line)))
```

```
sbcl -script script.lisp < datafile1 > datafile2
```

A semicolon starts a comment, which runs to the end of the line. By convention one semicolon is used for a comment after code on the same line, two for a comment on its own line inside a definition, and three for a comment at the top level of a file.

With `-script`, SBCL skips your init file, so Quicklisp is not loaded and neither is ASDF. A script that needs them starts by loading them itself.

2.7.2 Building an Executable

Because a running Lisp is an image in memory, most implementations can write that image to disk, and can write it as a program that starts in a function of your choosing. This is how a Lisp application is delivered to someone who does not have Lisp installed. In SBCL:

```
(defun main ()
  (format t "Hello from a standalone program!~%"))

(sb-ext:save-lisp-and-die "hello-app"
                          :toplevel #'main
                          :executable t)
```

This writes a file `hello-app` that runs on any machine of the same kind. It is about forty megabytes, because it contains the whole Lisp, compiler included. Each implementation has its own function for this. If you need one build recipe for several implementations, ASDF can do it; look up `program-op` in its manual.

2.8 Debugging in CL

Common Lisp has one of the best debugging and error-handling arrangements of any language, and it is always on. There is no separate debug mode and no debug build.

2.8.1 The Debugger

When an error happens in most languages, the program unwinds to the top and dies, and you read a stack trace to work out what the state must have been. When an error happens in Common Lisp, the program *stops where the error happened*, with everything still in place, and you are put into the debugger:

```
* (defun oops (x) (/ 10 x))
OOPS
* (oops 0)
```

```
debugger invoked on a DIVISION-BY-ZERO in thread
#<THREAD tid=59 "main thread" RUNNING {1200038003}>:
  arithmetic error DIVISION-BY-ZERO signalled
Operation was (/ 10 0).
```

Type `HELP` for debugger help, or `(SB-EXT:EXIT)` to exit from SBCL.

```
restarts (invokable by number or by possibly-abbreviated name):
  0: [ABORT] Exit debugger, returning to top level.
```

```
(SB-KERNEL::INTEGER-/INTEGER 10 0)
0]
```

The prompt `0]` is the debugger's. It is still a Lisp prompt: you can evaluate any expression there, look at variables, and even redefine the function that failed. `backtrace` shows the chain of calls that led here. Typing `0`, or `abort`, takes the restart numbered `0` and returns you to the normal prompt.

In SLIME or Sly the same information appears in its own window. Press the number of a restart to choose it, `q` to abort back to the REPL, `Return` on a line of the backtrace to see that call's local variables, and `v` to jump to its source.

2.8.2 Restarts

The list headed “restarts” is the unusual part. A restart is a way of carrying on that the program itself has offered. Try opening a file that does not exist:

```
* (with-open-file (in "/tmp/nope.conf") (read in))
```

```
debugger invoked on a FILE-DOES-NOT-EXIST in thread
#<THREAD tid=68 "main thread" RUNNING {1200038003}>:
  The file #P"/tmp/nope.conf" does not exist: No such file or directory
```

```
restarts (invokable by number or by possibly-abbreviated name):
  0: [CREATE   ] Reopen with :if-does-not-exist :create
  1: [CONTINUE ] Retry opening.
  2: [USE-VALUE] Try opening a different file.
  3: [ABORT    ] Exit debugger, returning to top level.
```

You could now create the file from another window and choose restart 1, and the program would carry on from the middle as if nothing had gone wrong. When the failure comes two hours into a long computation, that is worth a great deal. Section 3.8 shows how a program handles errors itself, without the debugger.

2.8.3 Stopping a Program Yourself

To interrupt a computation that is taking too long, press Control-C in the terminal, or **C-c** in the SLIME REPL. You land in the debugger, where you can look around and then either continue or abort.

To make a program stop at a particular place, call **break** there:

```
(defun risky (n)
  (break "n is ~a" n)
  (* n 2))
```

Calling `(risky 21)` enters the debugger with the message “n is 21” and a **CONTINUE** restart, which resumes the function and returns 42.

2.8.4 Trace, Describe and Friends

Trace prints every call to a function and what it returned, without your editing the function. It is especially good for seeing how a recursive function works:

```
CL-USER> (defun our-length (list)
           (if (null list)
               0
               (+ (our-length (rest list)) 1)))
OUR-LENGTH
CL-USER> (trace our-length)
(OUR-LENGTH)
CL-USER> (our-length '(a b))
0: (OUR-LENGTH (A B))
1: (OUR-LENGTH (B))
2: (OUR-LENGTH NIL)
2: OUR-LENGTH returned 0
1: OUR-LENGTH returned 1
0: OUR-LENGTH returned 2
2
CL-USER> (untrace our-length)
T
```

The running Lisp can also tell you about itself. **Describe** prints what is known about any object, **documentation** returns a documentation string, and **apropos** finds names containing a piece of text:

```
CL-USER> (documentation 'mapcar 'function)
"Apply FUNCTION to successive tuples of elements of LIST and MORE-LISTS.
Return list of FUNCTION return values."
```

```
CL-USER> (apropos "HASH-TABLE-" :cl)
HASH-TABLE-COUNT (fbound)
HASH-TABLE-P (fbound)
HASH-TABLE-REHASH-SIZE (fbound)
HASH-TABLE-REHASH-THRESHOLD (fbound)
HASH-TABLE-SIZE (fbound)
HASH-TABLE-TEST (fbound)
WITH-HASH-TABLE-ITERATOR (fbound)
```

2.8.5 Timing

Wrap any expression in the `time` macro to see how long it takes and how much memory it allocates. The expression’s value is returned as usual:

```
CL-USER> (defun fib (n)
           (if (< n 2)
               n
               (+ (fib (- n 1)) (fib (- n 2))))))
FIB
CL-USER> (time (fib 30))
Evaluation took:
  0.023 seconds of real time
  0.021844 seconds of total run time (0.021844 user, 0.000000 system)
  95.65% CPU
  65,398,250 processor cycles
  0 bytes consed

832040
```

“Consed” means allocated; Section 3.2.10 explains the word. For finding out where a larger program spends its time, the implementations provide profilers. SBCL’s is called `sb-sprof`.

2.9 Working with an AI Agent

Many programmers now work with an AI coding agent: a language model that reads the project, proposes changes, and runs commands. Lisp suits this way of working unusually well, for the same reasons it suits a person at a REPL.

An agent working in most languages writes code, runs the whole program or its tests, and reads the output. An agent connected to a running Lisp image can do what you do: evaluate one expression and look at the answer, ask the image for a function’s argument list or documentation instead of guessing, redefine one function and try it, and inspect the state left behind by an error. Mistakes are caught an expression at a time. The connection is usually made with the Model Context Protocol (MCP), a common way of giving a model tools to call. One open-source bridge between MCP and a running Lisp is `lisply-mcp`, <https://github.com/genworks/lisply-mcp>.

Some advice that has held up:

- Have the agent evaluate what it writes, in the image, before it shows it to you. Code that has only been written is a guess.
- Read what it produces. To check an agent's Lisp you need what this book teaches: what gets evaluated and when, what a function returns, what is modified in place.
- Let it work in a Lisp that is safe to break, such as a container, and keep your own work in version control.

An agent makes it more useful to understand the language, not less. You are the one who decides whether the program is right.

Chapter 3

The CL Language

This chapter is a condensed tour of the language. It covers what you need in order to read and write ordinary Common Lisp, and it leaves a good deal out. Chapter 5 lists longer books and the reference material for when you want the rest.

Type the examples in as you go. The text after the `CL-USER>` prompt is what you type, and the lines after it are what Lisp prints.

3.1 Overview of CL and its Syntax

The first thing to notice about CL is that it uses prefix notation. The most common thing to do, in a program or at the REPL, is to call a function, and you do that by writing a list: the name of the function first, then its arguments, all inside parentheses.

```
CL-USER> (+ 2 2)
4
```

This expression is the function named by the symbol `+`, followed by the arguments 2 and 2. Evaluating it returns 4.

3.1.1 How Expressions Are Evaluated

When CL is asked to evaluate an expression, it follows a few rules:

1. A number evaluates to itself:

```
CL-USER> 99
99
```

2. A string, which is text in double quotes, evaluates to itself:

```
CL-USER> "Be braver - you can't cross a chasm in two small jumps."
"Be braver - you can't cross a chasm in two small jumps."
```

3. A symbol, which is a bare name, is treated as a variable and evaluates to the variable's value. Variables come in Section 3.1.6.
4. A list is treated as a call. The first element names a function (or a macro, which we will come to). The remaining elements are the arguments. Each argument is evaluated, left to right, and the function is then called with the resulting values:

```
CL-USER> (expt 2 5)
32
```

Since the arguments are themselves expressions, calls nest to any depth:

```
CL-USER> (+ (+ 2 2) (+ 3 3))
10
```

Try it: Type these at your prompt and convince yourself that each result makes sense:

```
(+ 2)

(+)

(user-homedir-pathname)

(get-universal-time)      ; seconds since the start of the year 1900

(search "Dr." "I am Dr. Strangelove")

(subseq "I am Dr. Strangelove"
  (search "Dr." "I am Dr. Strangelove"))
```

Do not worry about getting lost among the parentheses. Your editor indents the code to show its structure and shows you which parentheses match, and after a short while you read Lisp by its indentation, as you read any other language.

3.1.2 Lisp Syntax Simplicity

What you have just read is nearly the whole syntax of the language. The one thing to add is that some operators are *macros* or *special operators* instead of functions. They are written the same way, with the operator's name first, but they do not necessarily evaluate all their arguments; each has its own rule. You will meet the common ones in this chapter and their rules are easy to remember.

If you come from a language such as C, Java, JavaScript or Python, you have been using a mixture of notations: infix for arithmetic ($2 + 2$), postfix for a few things (`i++`), and prefix for everything else (`max(a, b)`). Lisp uses the prefix form for everything and moves the opening parenthesis to the left of the function's name. That is the whole difference. In return there are no precedence rules to remember, and `+` can take any number of arguments.

3.1.3 Turning Off Evaluation

Sometimes you want an expression itself and not its value. The special operator `quote` returns its argument without evaluating it:

```
CL-USER> (quote a)
A
CL-USER> (quote (+ 1 2))
(+ 1 2)
```

The first returns the symbol `A` itself, where a bare `a` would have been looked up as a variable. The second returns a list of three items, where the unquoted form would have been a call returning 3.

This is needed so often that there is an abbreviation: a single quote character in front of an expression.

```
CL-USER> 'a
A
CL-USER> '(+ 1 2)
(+ 1 2)
```

Notice that Lisp printed the symbol in upper case. By default the reader converts the names of symbols to upper case as it reads them, so `a`, `A`, `hello` and `Hello` in your source all name symbols with upper-case names. People write their code in lower case and Lisp answers in capitals.

3.1.4 Fundamental CL Data Types

Common Lisp has the data types you know from other languages, such as numbers, strings and arrays, and some that may be new, such as symbols and lists. This section introduces numbers, strings, symbols and lists. The others come later in the chapter.

CL is dynamically typed: values have types, and variables can hold a value of any type. The type of a value is checked when the program runs. You may declare types where you want the compiler to check or optimize something, and you never have to.

Numbers

Numbers in CL form a family of types: integers, ratios, floating-point numbers and complex numbers. Most of the time you just think “number.” The arithmetic functions take any mixture and return a result of the appropriate kind.

```
CL-USER> (/ 1 3)
1/3
CL-USER> (+ 1/3 2/3)
1
CL-USER> (/ 1.0 3)
0.33333334
CL-USER> (expt 2 100)
1267650600228229401496703205376
CL-USER> (sqrt -4)
#C(0.0 2.0)
```

Dividing two integers gives an exact ratio, so one third is really one third, and does not get rounded until you ask for a floating-point number. Integers have no maximum size, so arithmetic on them never overflows.

Strings

A string is a sequence of characters written between double quotes. Strings can hold any Unicode text in all current implementations.

```
CL-USER> (string-upcase "hello")
"HELLO"
CL-USER> (concatenate 'string "Hello, " "World")
"Hello, World"
```

A single character is written with the prefix `#\`, as in `#\a` for the letter a and `#\Space` for a space.

Symbols

Symbols are names. You have already used some: `+`, `expt` and `search` are symbols that name functions. Symbols are also data in their own right. A program can hold a symbol in a variable, put it in a list, and compare it with another symbol, which is very fast because two occurrences of the same name are the very same object.

A symbol can name a function and a variable at the same time without any conflict. `List`, for example, is a built-in function, and you are free to use `list` as a variable name as well. There is no need to shorten it to `lst`.

To get a symbol itself where Lisp would otherwise evaluate it, quote it, as in Section 3.1.3:

```
CL-USER> 'my-symbol
MY-SYMBOL
```

Symbols whose names begin with a colon are *keywords*. A keyword evaluates to itself, so it needs no quote. Keywords are used as labels and option names throughout the language:

```
CL-USER> :red
:RED
```

Lists

A list is written as items between parentheses. The items can be of any type, including other lists. Since an unquoted list is a call, you quote a list when you mean it as data:

```
CL-USER> '(a b c)
(A B C)
CL-USER> '(a (b c) d)
(A (B C) D)
```

The quote covers everything inside the list.
Now look at this list:

```
(defun hello () (write-string "Hello, World!"))
```

It is also a program: the function definition from Chapter 2. Take a moment to see that it meets the description of a list. It has four items. The first two are symbols, the third is an empty list, and the fourth is a list of a symbol and a string.

Every CL program is made of lists like this one. That is why it is straightforward for a CL program to read, build and transform other CL programs, and it is the main thing that sets Lisp apart from other languages.

Besides quoting, you can make a list by calling the function `list`, which evaluates its arguments, as every function does, and returns a list of the results:

```
CL-USER> (list 'a 'b (+ 2 2))
(A B 4)
```

3.1.5 Functions

You define a named function with `defun`. It takes a name, a list of parameters, and a body of one or more expressions. The value of the last expression in the body is the value the function returns; there is no `return` statement.

```
CL-USER> (defun my-first-lisp-function ()
           (list 'hello 'world))
MY-FIRST-LISP-FUNCTION
CL-USER> (my-first-lisp-function)
(HELLO WORLD)
```

`Defun` is a macro, so it does not evaluate its arguments in the ordinary way: the name and the parameter list are taken as written, with no quoting.

If the first thing in the body is a string, it is kept as the function's documentation:

```
CL-USER> (defun square (x)
           "Return X multiplied by itself."
           (* x x))
SQUARE
CL-USER> (square 4)
16
CL-USER> (documentation 'square 'function)
"Return X multiplied by itself."
```

Names in Lisp are conventionally written in lower case with hyphens between the words.

3.1.6 Global and Local Variables

Global Variables

Global variables are established with `defparameter` or `defvar`:

```
CL-USER> (defparameter *todays-temp* 90)
*TODAYS-TEMP*
CL-USER> (defvar *todays-humidity* 70)
*TODAYS-HUMIDITY*
```

The asterisks are part of the names. They mean nothing to Lisp. It is a firm convention to name global variables this way so that they stand out, and you will see why it matters in a moment.

The two differ in one way. If the variable already has a value, `defvar` leaves it alone, and `defparameter` replaces it:

```
CL-USER> (defparameter *todays-temp* 100)
*TODAYS-TEMP*
CL-USER> (defvar *todays-humidity* 50)
*TODAYS-HUMIDITY*
CL-USER> *todays-temp*
100
CL-USER> *todays-humidity*
70
```

This matters because you reload files often while you work. Use `defvar` for state that should survive a reload, such as a table your program has been filling in, and `defparameter` for settings that should follow the source file.

To change the value of a variable, use `setf`:

```
CL-USER> (setf *todays-humidity* 30)
30
CL-USER> *todays-humidity*
30
```

You will also see `setq` in older code. It does the same thing for variables. `Setf` is more general, as later sections show, so it is the one to learn.

Local Variables

Local variables are introduced with `let`:

```
CL-USER> (let ((a 20)
               (b 30))
          (+ a b))
50
```

`Let` takes a list of bindings and then a body. Each binding is a pair: a variable name and an expression for its initial value. The body is one or more expressions, which can use the variables, and the value of the last one is the value of the whole `let`. Outside the `let`, the variables do not exist.

The initial values in a `let` are all computed before any of the variables exists, so one binding cannot refer to another. When you want that, use `let*`, which makes the bindings one after another:

```
CL-USER> (let* ((a 20)
                (b (* a 2)))
           (+ a b))
60
```

Variables made by `let` have *lexical scope*: they can be seen only by the code written inside the body of the `let`. You can tell where a variable is visible by looking at the text of the program.

Dynamic Scope

Global variables behave differently, and usefully so. If you bind a global variable with `let`, the new value is seen not only by the code inside the `let` but by every function called from there, for as long as the `let` is running. Afterwards the old value is back.

```
CL-USER> (defun report-humidity ()
           (format nil "Humidity is ~a%" *todays-humidity*))
REPORT-HUMIDITY
CL-USER> (report-humidity)
"Humidity is 30%"
CL-USER> (let ((*todays-humidity* 50))
           (report-humidity))
"Humidity is 50%"
CL-USER> (report-humidity)
"Humidity is 30%"
```

This is called *dynamic scope*, and global variables are also known as *special variables* because of it. It is the clean way to change a setting for the length of one operation. You do not have to remember to put the old value back, even if an error interrupts the work, and in a program with several threads the temporary value is seen only by the thread that made it. The standard output stream, for instance, is a special variable, so you can redirect the output of a whole body of code by binding it.

It is also why the asterisks matter: they tell you at a glance that a `let` is making a dynamic binding and not a lexical one.

3.2 The List as a Data Structure

This section presents the basic operators for working with lists.

3.2.1 Accessing the Elements of a List

The functions `first` through `tenth` return the elements of a list by position, and `nth` takes the position as a number, counting from zero:

```
CL-USER> (first '(a b c))
A
CL-USER> (second '(a b c))
B
CL-USER> (nth 0 '(a b c))
A
CL-USER> (nth 2 '(a b c))
C
```

3.2.2 The “Rest” of the Story

A very common pattern is to do something with the first element of a list and then do the same with what remains. The function `rest` returns the list without its first element:

```
CL-USER> (rest '(a b c))
(B C)
```

In older code, and in a good deal of current code, you will see `first` and `rest` written as `car` and `cdr`, names that survive from the machine Lisp was first written for. They mean the same thing.

3.2.3 The Empty List

The symbol `nil` is the empty list, which can also be written `()`. `Nil` evaluates to itself, so it needs no quote:

```
CL-USER> nil
NIL
CL-USER> ()
NIL
```

`Nil` is also what Lisp uses for “false.” The symbol `t` is the standard value for “true,” but in any test, *every value other than `nil` counts as true*. Many functions take advantage of this, as you will see.

The function `null` tests for the empty list:

```
CL-USER> (null '(a b c))
NIL
CL-USER> (null nil)
T
```

3.2.4 Are You a List?

`Listp` tests whether something is a list. Functions that ask a yes-or-no question are called *predicates*, and by convention their names end in `p`:

```
CL-USER> (listp '(pontiac cadillac chevrolet))
T
CL-USER> (listp 99)
NIL
CL-USER> (listp nil)
T
```

The last result is `T` because `nil` is a list: the empty one.

3.2.5 The Conditional If

Before going on with lists we need a way to make a choice. `if` takes three expressions: a test, a “then” and an “else.” It evaluates the test. If the result is true, it evaluates the second expression and returns its value; otherwise it evaluates the third and returns that. Only one of the two is ever evaluated, which is why `if` cannot be a function.

```
CL-USER> (if (> 3 4)
             "yes"
             "no")
"no"
CL-USER> (if (listp '("Chicago" "Detroit" "Toronto"))
             "it is a list"
             "it is not a list")
"it is a list"
```

Notice that `if` returns a value. In Lisp there is no division between statements and expressions: everything returns a value and can be used inside something else.

3.2.6 Length of a List

`length` returns the number of elements in a list:

```
CL-USER> (length '(gm ford stellantis volkswagen))
4
CL-USER> (length nil)
0
```

Like most of Common Lisp, `length` could be written in Common Lisp. Here is a simple version:

```
(defun our-length (list)
  (if (null list)
      0
      (+ (our-length (rest list)) 1)))
```

In English: the length of an empty list is zero, and the length of any other list is one more than the length of its rest. A function that calls itself like this is *recursive*, and recursion is a natural way to express operations on lists.

3.2.7 Member of a List

`member` tells you whether an item is in a list. If it is not, `member` returns `nil`. If it is, `member` returns the part of the list that starts with the item:

```
CL-USER> (member 'dallas '(boston san-francisco portland))
NIL
CL-USER> (member 'san-francisco '(boston san-francisco portland))
(SAN-FRANCISCO PORTLAND)
```

Because anything that is not `nil` counts as true, the second result works perfectly well as a “yes” in a test, and it carries some useful information besides. This is a common idiom in Lisp.

3.2.8 Getting Part of a List

`Subseq` returns part of a list. It takes the list, the position to start at, and optionally the position to stop before:

```
CL-USER> (subseq '(a b c d) 1 3)
(B C)
CL-USER> (subseq '(a b c d) 1)
(B C D)
```

`Subseq`, `length` and many other functions work on any *sequence*, which means lists, strings and the vectors of Section 3.6.2. You saw `subseq` applied to a string at the start of the chapter.

3.2.9 Appending Lists

`Append` takes any number of lists and returns a new list made of all their elements:

```
CL-USER> (defparameter *slides* '(introduction welcome lists functions))
*SLIDES*
CL-USER> (append *slides* '(numbers))
(INTRODUCTION WELCOME LISTS FUNCTIONS NUMBERS)
CL-USER> *slides*
(INTRODUCTION WELCOME LISTS FUNCTIONS)
```

`Append` did not change `*slides*`. Like most CL functions, it returns a new value and leaves its arguments alone. If you want the variable to change, say so:

```
CL-USER> (setf *slides* (append *slides* '(numbers)))
(INTRODUCTION WELCOME LISTS FUNCTIONS NUMBERS)
```

3.2.10 Adding Elements to a List

`Cons` adds one element to the front of a list:

```
CL-USER> (cons 'a '(b c d))
(A B C D)
```

`Cons` is the primitive from which lists are built. A list is a chain of pairs, called *cons cells*, each holding one element and a pointer to the next pair. `Cons` makes one new cell that points at the existing list, so it is fast, and the old list is not copied or changed. When you hear that a program “conses a lot,” it means that the program allocates a lot of memory, which the garbage collector will later have to reclaim.

The macros `push` and `pop` combine `cons` and `rest` with updating a variable, which lets you use a list as a stack:

```
CL-USER> (defparameter *stack* '(b c))
*STACK*
CL-USER> (push 'a *stack*)
(A B C)
```

```
CL-USER> (pop *stack*)
A
CL-USER> *stack*
(B C)
```

3.2.11 Removing Elements from a List

`Remove` returns a new list without the occurrences of an item. `Remove-if` and `remove-if-not` take a predicate in place of the item:

```
CL-USER> (remove 1000 '(1 2 3 1000 4))
(1 2 3 4)
CL-USER> (remove-if-not #'evenp '(1 2 3 4 5 6))
(2 4 6)
```

The notation `#'evenp` means “the function named `evenp`.” Section 3.4 explains it.

3.2.12 Sorting Lists

`Sort` sorts a sequence, given a function that says whether one element should come before another:

```
CL-USER> (defparameter *numbers* (list 1 3 5 7 2 4 6))
*NUMBERS*
CL-USER> (setf *numbers* (sort *numbers* #'<))
(1 2 3 4 5 6 7)
CL-USER> (setf *numbers* (sort *numbers* #'>))
(7 6 5 4 3 2 1)
```

`Sort` is an exception to the rule that functions leave their arguments alone. For speed it is allowed to reuse the cells of the list you give it, so afterwards the original list is in an undefined state. Always use the value that `sort` returns, as above. To keep the original, `sort` a copy:

```
(defun safe-sort (list predicate)
  (sort (copy-list list) predicate))
```

Functions that may modify their arguments are called *destructive*. There are only a few common ones, and their documentation says so. Never use one on a quoted list written in your source, because that list is a constant that belongs to the program.

3.2.13 Treating a List as a Set

`Union`, `intersection` and `set-difference` treat two lists as sets. Sets have no order, so do not rely on the order of the results:

```
CL-USER> (union '(1 2 3) '(2 3 4))
(4 1 2 3)
CL-USER> (intersection '(1 2 3) '(2 3 4))
(3 2)
CL-USER> (set-difference '(1 2 3 4) '(2 3))
(4 1)
```

3.2.14 Mapping a Function to a List

When you have a list and want another list of the same length, with each element transformed, use `mapcar`. It takes a function and a list, calls the function on each element, and returns a list of the results:

```
CL-USER> (defun twice (num)
           (* num 2))
TWICE
CL-USER> (mapcar #'twice '(1 2 3 4))
(2 4 6 8)
```

Given several lists, `mapcar` takes one argument from each:

```
CL-USER> (mapcar #' + '(1 2 3) '(10 20 30))
(11 22 33)
```

Reduce combines all the elements of a list into one value, using a function of two arguments:

```
CL-USER> (reduce #' + '(1 2 3 4))
10
```

`Mapcar`, `remove-if-not` and `reduce` are the map, filter and reduce of other languages.

3.2.15 Property Lists

A property list, or *plist*, is a simple way to keep named values. It is a list in which the odd-numbered items are names, usually keywords, and each is followed by its value. `Getf` looks up a name:

```
CL-USER> (getf '(:michigan "Lansing"
                 :illinois "Springfield"
                 :pennsylvania "Harrisburg")
            :illinois)
"Springfield"
```

Plists are fine for a handful of items. For large collections, use the hash tables of Section 3.6.1.

3.3 Control of Execution

Controlling execution in CL comes down to controlling which expressions get evaluated.

3.3.1 If, When and Unless

If takes exactly one “then” expression and one “else” expression. To do several things in either branch, group them with **progn**, which evaluates its arguments in order and returns the value of the last:

```
(if (eql status :all-systems-go)
    (progn (broadcast-countdown)
           (launch-rocket))
    (progn (broadcast-apology)
           (shut-down-rocket)))
```

When there is no “else,” use **when**. It takes a test and any number of expressions, with no need for **progn**. **Unless** is the opposite: it evaluates its body if the test is false.

```
CL-USER> (when (> 4 3)
           (print "four is more")
           :done)
```

```
"four is more"
:DONE
```

```
CL-USER> (unless (> 4 3)
           :never)
```

```
NIL
```

3.3.2 Logical Operators

And evaluates its arguments from left to right. It stops and returns **nil** as soon as one of them is false. If none is, it returns the value of the last one. **Or** stops at the first argument that is true and returns that value:

```
CL-USER> (and (listp '(chicago detroit boston new-york))
              (listp 3.1415))
```

```
NIL
```

```
CL-USER> (or (listp '(chicago detroit boston new-york))
              (listp 3.1415))
```

```
T
```

```
CL-USER> (or nil 42 99)
```

```
42
```

The last example shows a common way to supply a default: **(or value default)** returns **value** unless it is **nil**.

Not returns **t** if its argument is **nil**, and **nil** otherwise.

3.3.3 Cond

For a chain of tests, where another language would say “if ... else if ... else,” use **cond**. Each clause is a list holding a test followed by expressions. **Cond** tries the tests in order, and at the first true one it evaluates that clause’s expressions and returns. A final clause whose test is **t** catches everything else:

```
CL-USER> (let ((n 4))
           (cond ((> n 10) "It's a lot")
                 ((= n 10) "It's kind of a lot")
                 (t "It's not a lot")))
"It's not a lot"
```

With `cond` we can write our own version of `member`, very close to how you would describe it in English:

```
(defun our-member (elem list)
  (cond ((null list) nil)
        ((eql elem (first list)) list)
        (t (our-member elem (rest list)))))
```

“If the list is empty, the answer is no. If the item is the same as the first element, return the list. Otherwise look in the rest.”

3.3.4 Case

To choose among fixed values, `case` is more concise than `cond`:

```
CL-USER> (let ((color :red))
           (case color
             (:blue "Blue is okay")
             (:red "Red is actually her favorite color")
             (otherwise "We have never heard of that!")))
"Red is actually her favorite color"
```

`Case` compares with `eql`, so it works with symbols, keywords, integers and characters, and not with strings. Section 3.10.4 says why. `Ecase` is like `case` but signals an error if nothing matches, which is what you want when the list of cases is supposed to be complete.

3.3.5 Iteration

Lisp has recursion and mapping, and it also has ordinary loops. `Dolist` runs its body once for each element of a list, and `dotimes` runs it a given number of times, counting from zero:

```
CL-USER> (dolist (city '("Detroit" "Delft" "Tokyo"))
           (format t "Hello, ~a!~%" city))
Hello, Detroit!
Hello, Delft!
Hello, Tokyo!
NIL
CL-USER> (let ((result 0))
           (dotimes (n 10)
             (setf result (+ result n)))
           result)
```

45

The `loop` macro is a small language of its own for iteration. It reads almost like English, and it can collect, sum, count and filter as it goes:

```
CL-USER> (loop for n from 1 to 5 collect (* n n))
(1 4 9 16 25)
CL-USER> (loop for elem in '(4 5 6 7) sum elem)
22
CL-USER> (loop for n from 1 to 10
           when (evenp n) collect n)
(2 4 6 8 10)
```

Loop can do a great deal more. The references in Chapter 5 describe it in full. To leave any of these loops early, call **return**.

3.4 Functions as Objects

3.4.1 Named Functions

A function in CL is an object, like a number or a list. It can be stored in a variable, passed to another function, and returned as a result. To get the function object that a name refers to, write **#'** before the name:

```
CL-USER> #' +
#<FUNCTION +>
```

A function object has no printed form that could be read back in, so Lisp prints a description between **#<** and **>**. You will see this notation for many kinds of object.

To call a function object, use **funcall**, giving the arguments one by one, or **apply**, giving them as a list:

```
CL-USER> (funcall #' + 1 2)
3
CL-USER> (apply #' + '(1 2 3))
6
```

3.4.2 Functional Arguments

You have already passed functions to **mapcar**, **sort**, **reduce** and **remove-if-not**. Functions that take other functions as arguments are called higher-order functions, and CL uses them everywhere. Your own functions qualify as soon as you define them:

```
CL-USER> (defun add-3 (num)
           (+ num 3))
ADD-3
CL-USER> (mapcar #'add-3 '(2 3 4))
(5 6 7)
```

3.4.3 Anonymous Functions

Often a function is needed in just one place, and it is not worth giving it a name. **Lambda** makes a function on the spot. It looks like **defun** without the name:

```
CL-USER> (mapcar (lambda (num) (+ num 3))
                '(2 3 4))
(5 6 7)
CL-USER> (sort (list '(4 "Buffy") '(2 "Keiko") '(1 "Judy") '(3 "Aruna"))
              (lambda (x y)
                (< (first x) (first y))))
((1 "Judy") (2 "Keiko") (3 "Aruna") (4 "Buffy"))
```

A function made by `lambda` can use the variables that are visible where it is written, and it keeps them for as long as it lives. Such a function is called a *closure*:

```
CL-USER> (defun make-adder (n)
            (lambda (x) (+ x n)))
MAKE-ADDER
CL-USER> (defparameter *add-10* (make-adder 10))
*ADD-10*
CL-USER> (funcall *add-10* 5)
15
CL-USER> (mapcar (make-adder 100) '(1 2 3))
(101 102 103)
```

Each call to `make-adder` returns a new function that remembers its own `n`.

3.4.4 Optional Arguments

Parameters after the symbol `&optional` in a parameter list may be left out by the caller. Each can be given a default value; without one, the default is `nil`:

```
CL-USER> (defun greeting-list (&optional (username "Jake"))
            (list 'hello 'there username))
GREETING-LIST
CL-USER> (greeting-list)
(HELLO THERE "Jake")
CL-USER> (greeting-list "Joe")
(HELLO THERE "Joe")
```

3.4.5 Keyword Arguments

Parameters after `&key` are optional too, and the caller passes them by name, in any order. They are written the same way:

```
(defun greeting-list (&key (username "Jake")
                       (greeting "how are you?"))
  (list 'hello 'there username greeting))
```

and called by putting a keyword before each value:

```
CL-USER> (greeting-list :greeting "how have you been?")
(HELLO THERE "Jake" "how have you been?")
CL-USER> (greeting-list :greeting "how have you been?" :username "Joe")
(HELLO THERE "Joe" "how have you been?")
```

Keyword arguments make calls easy to read, and they let you add an option to a function later without disturbing its existing callers. The built-in functions use them heavily. `Sort`, for instance, takes a `:key` function that picks out the part of each element to compare, so the sort above could have been written (`sort people #'< :key #'first`).

A parameter after `&rest` receives all the remaining arguments as a list, which is how you write a function that takes any number of them:

```
CL-USER> (defun average (&rest numbers)
           (/ (reduce #'+ numbers) (length numbers)))
AVERAGE
CL-USER> (average 1 2 3 4)
5/2
```

3.4.6 Multiple Values

A function can return more than one value. `Floor`, for example, divides and returns both the quotient and the remainder:

```
CL-USER> (floor 7 2)
3
1
```

If you use the call in the ordinary way, you get the first value and the others are quietly dropped, so `(+ (floor 7 2) 10)` is 13. To receive them all, use `multiple-value-bind`:

```
CL-USER> (multiple-value-bind (quotient remainder)
           (floor 7 2)
           (list quotient remainder))
(3 1)
```

Your own functions return several values by calling `values`: `(values 3 1)`.

3.5 Input, Output, Streams, and Strings

Input and output in CL go through *streams*. A stream is a source or destination of data: the terminal, a file, a network connection, or a string in memory. Two special variables hold the defaults: `*standard-input*` and `*standard-output*`.

3.5.1 Format

`Format` is the function you will use for most output. It takes a destination, a control string, and arguments that are filled in where the control string has directives. Directives start with a tilde:

```
CL-USER> (format t "Hello there, ~a!~%" 'bob)
Hello there, BOB!
NIL
```

The destination `t` means standard output. The directive `~a` prints the next argument in a form meant for people, and `~%` starts a new line. `Format` printed its text and then returned `nil`, which the REPL showed.

If the destination is `nil`, `format` prints nothing and returns the text as a string. This is the usual way to build a string from pieces:

```
CL-USER> (format nil "Hello there, ~a!" 'bob)
"Hello there, BOB!"
```

A few more directives will cover most of what you need: `~s` prints an argument the way you would type it in a program, `~d` prints an integer, `~,2f` prints a number with two decimal places, and `~{ ... ~}` repeats over a list:

```
CL-USER> (format nil "~a and ~s" "a string" "a string")
"a string and \"a string\""
CL-USER> (format nil "~d items at ~,2f each" 3 4.5)
"3 items at 4.50 each"
CL-USER> (format nil "~{~a~, ~}" '(1 2 3))
"1, 2, 3"
```

There are many others. One prints numbers in English and another in Roman numerals:

```
CL-USER> (format nil "~r" 2026)
"two thousand twenty-six"
CL-USER> (format nil "~@r" 2026)
"MMXXVI"
```

3.5.2 Print and Princ

`Print` outputs one object in the form that Lisp could read back in, on a new line, and returns the object. `Princ` outputs it in the form meant for people. The difference shows with strings:

```
CL-USER> (print "hello")

"hello"
"hello"
CL-USER> (princ "hello")
hello
"hello"
```

In each case the first thing you see is the output and the second is the return value, printed by the REPL. `Print` returns its argument, so you can wrap it around any expression in the middle of a program to see a value go by without changing what the program does.

3.5.3 Read

Read is the other half of the REPL: it takes text from a stream and returns the Lisp object that the text describes. It does not evaluate anything. The whole reader is available to your programs, which makes Lisp's own notation a ready-made data format.

```
CL-USER> (read-from-string "(a b (c 42) \"text\")")
(A B (C 42) "text")
19
```

(The second value is the position where reading stopped.) Only read data that you trust this way. For text from outside, read lines with **read-line**, which returns one line as a string, or use a library for a format such as JSON, as in Chapter 4.

3.5.4 File Input and Output

To use a file you open a stream to it. The macro **with-open-file** opens the file, gives you the stream in a variable for the length of its body, and closes the file when the body is finished, even if it is left because of an error.

To write a file:

```
CL-USER> (with-open-file (out "/tmp/readme.txt"
                           :direction :output
                           :if-exists :supersede)
          (format out "Hello there~%")
          (format out "Second line~%"))
NIL
```

and to read it back, a line at a time:

```
CL-USER> (with-open-file (in "/tmp/readme.txt")
          (loop for line = (read-line in nil)
                while line
                collect line))
("Hello there" "Second line")
```

Opening for input is the default, so the second form needs no options. The **nil** given to **read-line** tells it to return **nil** at the end of the file, where it would otherwise signal an error.

Probe-file tests whether a file exists.

3.5.5 Pathnames

You can name files with ordinary strings, as above. Lisp turns them into *pathname* objects, which you can take apart and combine without cutting strings up by hand:

```
CL-USER> (defparameter *my-path* (pathname "/tmp/readme.txt"))
*MY-PATH*
CL-USER> *my-path*
```

```
#P"/tmp/readme.txt"
CL-USER> (pathname-name *my-path*)
"readme"
CL-USER> (pathname-type *my-path*)
"txt"
CL-USER> (merge-pathnames "notes.txt" *my-path*)
#P"/tmp/notes.txt"
```

3.6 Hash Tables, Arrays, Structures, and Classes

Lists are convenient, and they are not the answer to everything. CL has the other data structures you would expect.

3.6.1 Hash Tables

A hash table maps keys to values, like a dictionary or map in other languages. Lookup stays fast however large the table grows. Make one with `make-hash-table`, look things up with `gethash`, and store them with `setf` of `gethash`:

```
CL-USER> (defparameter *ratings* (make-hash-table))
*RATINGS*
CL-USER> (setf (gethash :lisp *ratings*) :excellent)
:EXCELLENT
CL-USER> (gethash :lisp *ratings*)
:EXCELLENT
CL-USER> (gethash :cobol *ratings*)
NIL
CL-USER> (gethash :cobol *ratings*)
NIL
```

This is `setf` doing something `setq` cannot. Its first argument is not just a variable but a *place*: an expression that says where a value lives. To store into anything, you write the expression that would fetch the value and wrap it in `setf`. The same rule works for hash tables, arrays, objects and parts of lists, so there is no separate “put” function to learn for each.

`Gethash` returns two values. The second says whether the key was found, which lets you tell a missing key from a key whose value is `nil`.

A table made with no options compares keys with `eql`, which suits symbols and numbers. For string keys, ask for `equal`:

```
CL-USER> (defparameter *capitals* (make-hash-table :test #'equal))
*CAPITALS*
CL-USER> (setf (gethash "Michigan" *capitals*) "Lansing")
"Lansing"
CL-USER> (gethash "Michigan" *capitals*)
"Lansing"
CL-USER> (gethash "Michigan" *capitals*)
T
```

`Maphash` calls a function on each key and value, and `hash-table-count` returns the number of entries.

3.6.2 Arrays

Arrays hold values in a grid of one or more dimensions, indexed by integers from zero, with fast access to any element. `make-array` creates one and `aref` reads an element. As with hash tables, `setf` of the same expression stores one:

```
CL-USER> (defparameter *grid* (make-array '(3 3) :initial-element 0))
*GRID*
CL-USER> (setf (aref *grid* 0 0) "Dave")
"Dave"
CL-USER> (setf (aref *grid* 0 1) "John")
"John"
CL-USER> *grid*
#2A(("Dave" "John" 0) (0 0 0) (0 0 0))
```

A one-dimensional array is called a *vector*. Vectors can be written directly with `#( ... )`, and they can be made to grow:

```
CL-USER> (aref #(10 20 30) 1)
20
CL-USER> (defparameter *v* (make-array 0 :adjustable t :fill-pointer t))
*V*
CL-USER> (vector-push-extend :a *v*)
0
CL-USER> (vector-push-extend :b *v*)
1
CL-USER> *v*
#(:A :B)
```

Strings are vectors of characters. Vectors and lists together are the sequences, so `length`, `subseq`, `sort`, `remove`, `find`, `position` and `reduce` work on all of them.

3.6.3 Structures

A structure is a record with named slots. One `defstruct` form defines the type together with a function to make instances, a function to read each slot, and a predicate:

```
CL-USER> (defstruct point
          x
          y)
POINT
CL-USER> (defparameter *p* (make-point :x 3 :y 4))
*P*
CL-USER> (point-x *p*)
3
CL-USER> (setf (point-y *p*) 10)
10
CL-USER> *p*
#S(POINT :X 3 :Y 10)
```

Structures are simple and fast. When you need more than that, use classes.

3.6.4 Classes and Methods

The Common Lisp Object System, CLOS, is the object-oriented part of the language. A class describes a kind of object: its slots, and the classes it inherits from. `Defclass` defines one:

```
(defclass box ()
  ((length :initarg :length :initform 10 :accessor box-length)
   (width  :initarg :width  :initform 10 :accessor box-width)
   (height :initarg :height :initform 10 :accessor box-height)))
```

After the name comes the list of superclasses, empty here, and then the slots. For each slot, `:initarg` gives the keyword used to set it when an object is made, `:initform` gives its default value, and `:accessor` names a function for reading the slot, and for writing it with `setf`.

Behavior is defined with *generic functions* and *methods*. A generic function is called like any other function. What it does depends on the classes of its arguments, and each method supplies the behavior for one case:

```
(defgeneric volume (thing)
  (:documentation "Return the volume of THING."))

(defmethod volume ((self box))
  (* (box-length self)
     (box-width self)
     (box-height self)))
```

`Make-instance` creates an object:

```
CL-USER> (defparameter *box* (make-instance 'box :height 2))
*BOX*
CL-USER> (volume *box*)
200
CL-USER> (setf (box-width *box*) 5)
5
CL-USER> (volume *box*)
100
```

A subclass inherits slots and methods:

```
CL-USER> (defclass labeled-box (box)
  ((label :initarg :label :accessor box-label)))
#<STANDARD-CLASS COMMON-LISP-USER::LABELED-BOX>
CL-USER> (volume (make-instance 'labeled-box :label "Fragile"))
1000
```

If you know Java, C++ or Python, notice what is different here. Methods do not live inside classes. A call is written `(volume box)`, like any function call, and not `box.volume()`. One consequence is that you can add a method for a class you did not write, including the built-in ones, without touching it. Another is that a method can be chosen by the classes of *all* its arguments, not only the first:

```
CL-USER> (defgeneric add (a b))
#<STANDARD-GENERIC-FUNCTION COMMON-LISP-USER::ADD (0)>
CL-USER> (defmethod add ((a string) (b string))
           (concatenate 'string a b))
#<STANDARD-METHOD COMMON-LISP-USER::ADD (STRING STRING) {1201ACA633}>
CL-USER> (defmethod add ((a number) (b number))
           (+ a b))
#<STANDARD-METHOD COMMON-LISP-USER::ADD (NUMBER NUMBER) {1201B1BD13}>
CL-USER> (add "hello " "there")
"hello there"
CL-USER> (add 3 4)
7
```

CLOS goes a good deal further than this, with multiple inheritance, methods that run before, after and around others, and classes that can be redefined while instances of them exist. The existing objects are updated to match. It fits the way Lisp is developed, and it is one of the reasons a Lisp program can stay up while it is being changed.

3.7 Macros

You have been using macros since the first `defun`: `when`, `cond`, `dolist`, `loop`, `setf` and `with-open-file` are all macros. This section shows what they are and how to write a small one. Writing macros well is a subject for the longer books. Reading them is something you need from the first day.

A function receives the *values* of its arguments. A macro receives its arguments as unevaluated *code*, as the lists and symbols you typed, and returns a new piece of code to be used in their place. It is a function from code to code, run by the compiler. Because Lisp code is made of lists, a macro is written with the same list operations as any other program.

Suppose `unless` did not exist. A function could not do its job, because a function's arguments are all evaluated before it is called, and the point of `unless` is that the body sometimes must not be. A macro can:

```
(defmacro our-unless (test &body body)
  '(if ,test
      nil
      (progn ,@body)))
```

`Defmacro` looks like `defun`. The parameter `test` receives the first expression in the call, and `&body` collects the rest as a list. The body builds the replacement code with the backquote notation: a backquote is like a quote, except that inside it a comma means “put the value of this here,” and comma-at means “splice the elements of this list in here.” The result is a template with holes.

You can ask Lisp what a macro call turns into, with `macroexpand-1`:

```
CL-USER> (macroexpand-1 '(our-unless (> 3 4) (print "no") :done))
(IF (> 3 4)
  NIL
  (PRINT "no")
  DONE)
```

```
(PROGN (PRINT "no") :DONE))
T
```

and the new operator works like a built-in one:

```
CL-USER> (our-unless (> 3 4)
               (print "three is not more")
               :done)

"three is not more"
:DONE
```

The built-in macros are no different. In SBCL:

```
CL-USER> (macroexpand-1 '(when (> 4 3) :done))
(IF (> 4 3)
    :DONE)
T
```

When you meet a macro you do not know, expand it. Your editor has a key for this (Table 2.1).

This is what Chapter 1 meant by extending the language from the inside. The built-in `with-open-file` is a macro that someone wrote, and in the same way you can write your own, such as a `with-connection` or a `define-report`, for the needs of your program. A good rule for beginners is to write a function whenever a function will do, and to reach for a macro when you need to control evaluation or to remove a pattern that keeps repeating in your code.

3.8 Conditions and Error Handling

Chapter 2 showed what happens at the REPL when something goes wrong: you land in the debugger. A finished program usually has to deal with problems itself. In CL an error is represented by an object called a *condition*, and conditions are organized into classes, such as `division-by-zero`, `file-error` and `type-error`, all of which are kinds of `error`.

`Handler-case` runs an expression and says what to do if a condition of a given class is signaled inside it. It corresponds to `try` and `catch` in other languages:

```
CL-USER> (handler-case (/ 10 0)
           (division-by-zero ()
            :infinity))
:INFINITY
```

To signal an error of your own, call `error` with a message in the style of `format`:

```
(defun parse-age (string)
  (let ((age (parse-integer string)))
    (when (minusp age)
      (error "An age cannot be negative: ~a" age))
    age))
```

```
CL-USER> (parse-age "42")
42
CL-USER> (handler-case (parse-age "forty")
  (error (condition)
    (format t "Could not read the age: ~a-%" condition)
    nil)))
Could not read the age: junk in string "forty"
NIL
```

The handler names the class `error`, so it catches both the error that `parse-integer` signaled here and the one `parse-age` would signal for a negative number. The variable `condition` holds the condition object, and printing it gives its message.

To make sure something happens on the way out, whether or not there was an error, use `unwind-protect`. Its first expression is the work and the rest are the cleanup:

```
CL-USER> (unwind-protect
  (progn (print "working") :result)
  (print "cleaning up"))

"working"
"cleaning up"
:RESULT
```

This is what `with-open-file` uses to close the file.

The condition system goes further than try and catch in one important way. In most languages, by the time an error is caught, the code that ran into it has been abandoned. In CL the handler runs first, while that code is still waiting, and it can choose one of the restarts that the code has offered: use this value in place of the bad one, skip this record, try again. The low-level code knows the possible ways to recover and the high-level code decides which one to use. The restarts you saw in the debugger in Section 2.8 are the same mechanism with a person making the choice. The operators to look up when you need this are `handler-bind` and `restart-case`.

3.9 Packages

A package is a namespace for symbols. Packages keep the names in one library from colliding with the same names in another, and they let a library say which of its names are meant for others to use.

There is always a current package, held in the special variable `*package*`, and the REPL shows it in the prompt. A new Lisp starts in the package `COMMON-LISP-USER`, or `CL-USER` for short, which is a scratch area. A program of any size should have a package of its own. `Defpackage` defines one:

```
(defpackage :inventory
  (:use :common-lisp)
  (:export #:add-item #:*items*))
```

This says that code in the `inventory` package can use all the standard names of the language, which live in the package `COMMON-LISP`, and that two of its own names are for the outside world. (The `#:` prefix just writes a name without creating a symbol in the current package. You will also see keywords and strings used here.)

Each source file then begins by saying which package it is in:

```
(in-package :inventory)

(defvar *items* nil)

(defun add-item (item)
  (push item *items*))

(defun helper ()
  :internal)
```

From another package, an exported name is written with the package name and one colon:

```
CL-USER> (inventory:add-item :widget)
(:WIDGET)
CL-USER> inventory:*items*
(:WIDGET)
```

A name that was not exported can still be reached by using two colons, as in `inventory::helper`. The second colon is a warning to yourself: this name is not part of the library's interface and may change. It is handy for poking around at the REPL and has no place in finished code.

If you are in a different package from the one you think you are in, Lisp will tell you that functions you have just defined do not exist. When that happens, look at the prompt.

Package names should be distinctive, since all packages share one list of names, and long names are tedious to type. A package can give another package a short local nickname for its own use:

```
(defpackage :report
  (:use :common-lisp)
  (:local-nicknames (:a :alexandria)))
```

after which code in `report` can write `a:flatten`. Local nicknames came after the standard, and every implementation mentioned in this book supports them.

3.9.1 The Keyword Package

Keywords, the symbols written with a leading colon, live in a package of their own named `KEYWORD`. That is why `:red` means the same thing wherever it is written, whatever the current package, and why keywords are used for option names and for labels in data.

3.10 Common Stumbling Blocks

This section goes over the things that most often trip up newcomers.

3.10.1 Quotes

A symbol without a quote is a variable. With a quote it is the symbol itself:

```
CL-USER> (defparameter *x* 1)
*X*
CL-USER> *x*
1
CL-USER> ' *x*
*X*
```

A list without a quote is a call. With a quote it is the list:

```
CL-USER> (first '(+ 1 2))
+
CL-USER> (first (+ 1 2))
; Evaluation aborted on #<SIMPLE-TYPE-ERROR expected-type: LIST datum: 3>.
```

In the second, `(+ 1 2)` was evaluated first, and `first` was handed the number 3. Numbers, strings and keywords evaluate to themselves and never need a quote.

3.10.2 Function Argument Lists

When you define a function, the parameters go in a list of their own. When you call it, the arguments follow the name directly:

```
CL-USER> (defun square (num)
           (* num num))
SQUARE
CL-USER> (square 4)
16
```

If you are used to writing `square(4)`, sooner or later you will type `(square (4))`. Lisp will try to call a function named 4 and complain of an illegal function call.

3.10.3 Symbols vs. Strings

The symbol `AAA` and the string `"AAA"` are different kinds of thing. A symbol is a name: there is exactly one symbol with a given name in a given package, and comparing two symbols is a single machine instruction. A string is a piece of text: two strings with the same characters are still two objects.

```
CL-USER> (eql 'aaa 'aaa)
T
CL-USER> (eql "bbb" "bbb")
NIL
CL-USER> (string= "bbb" "bbb")
T
```

Use symbols and keywords for things your program needs to tell apart, such as the states of a machine or the options of a function. Use strings for text that people read or that comes from outside. The string operations do not apply to symbols:

```
CL-USER> (defparameter *sentence* "Jane dates only Lisp programmers")
*SENTENCE*
CL-USER> (char *sentence* 3)
#\e
CL-USER> (subseq *sentence* 0 4)
"Jane"
CL-USER> (position #\Space *sentence*)
4
CL-USER> (subseq *sentence* 11)
"only Lisp programmers"
```

Remember also that the reader converts symbol names to upper case, so the name of the symbol you type as `hello` is the string `"HELLO"`.

3.10.4 Equality

CL has several functions for testing equality, because “the same” means different things for different kinds of object. A rule of thumb:

- `eq` asks whether two things are the very same object. Use it for symbols and keywords. It is also true for two integers of the same value and for two identical characters. It is the default test used by `member`, `case`, `remove`, `find` and hash tables.
- `=` compares numbers by value, whatever their types.
- `equal` compares structure: two lists with equal elements are `equal`, and so are two strings with the same characters.
- `string=` compares strings, and `string-equal` does so ignoring case.

```
CL-USER> (eq 3 3.0)
NIL
CL-USER> (= 3 3.0)
T
CL-USER> (eq '(a b c) (list 'a 'b 'c))
NIL
CL-USER> (equal '(a b c) (list 'a 'b 'c))
T
CL-USER> (string-equal "bbb" "BBB")
T
```

The functions that use `eq` by default take a `:test` argument to change it. Forgetting this with strings is the classic mistake:

```
CL-USER> (member "blue" '("red" "blue" "green" "yellow"))
NIL
CL-USER> (member "blue" '("red" "blue" "green" "yellow")
          :test #'string=)
("blue" "green" "yellow")
```

Floating-point numbers that ought to be equal often differ by a tiny amount, in Lisp as in every language. Compare them with a tolerance:

```
CL-USER> (= (+ 0.1d0 0.2d0) 0.3d0)
NIL
CL-USER> (defun nearly-equal-p (a b &optional (tolerance 1d-9))
          (< (abs (- a b)) tolerance))
NEARLY-EQUAL-P
CL-USER> (nearly-equal-p (+ 0.1d0 0.2d0) 0.3d0)
T
```

3.10.5 Distinguishing Macros from Functions

Macros and functions are called with the same syntax, and it rarely matters which is which. Assume a function unless you have a reason to think otherwise. Most macros are easy to spot:

- the familiar control operators, such as `when`, `cond`, `case`, `and`, `or` and `loop`;
- names that begin with `def`, such as `defun` and `defparameter`;
- names that begin with `with-`, such as `with-open-file`;
- names that begin with `do`, such as `dolist` and `dotimes`.

A small number of operators, including `if`, `let`, `quote`, `setq` and `progn`, are neither. They are the special operators that the language is built on, and they behave like macros as far as you are concerned.

If you are in doubt, ask the running Lisp:

```
CL-USER> (macro-function 'list)
NIL
CL-USER> (special-operator-p 'if)
T
```

`Macro-function` returns `nil` for anything that is not a macro. `Describe` will tell you all of this at once, and your editor's key for it is in Table 2.1.

3.10.6 Modifying What You Should Not

A quoted list in your source is a constant. Lisp is allowed to store one copy of it and share it, so changing it with a destructive operation such as `sort` can change your program in ways that are very hard to track down. If you are going to modify a list, build it with `list` or `copy-list`, as the `sort` examples in this chapter did.

Destructive operations exist for speed, and you seldom need them. A program that builds new lists and lets the garbage collector take the old ones is easier to get right, and usually fast enough. Write it that way first and measure with `time` before you change anything.

Chapter 4

Interfaces

A program has to deal with the world outside it: the operating system, code written in other languages, the network, databases, the web. The Common Lisp standard covers files and streams and stops there. The rest is done by libraries, and for each job there is a well-established portable one that works the same on every implementation. This chapter introduces the ones you are likely to need first.

Each is loaded with Quicklisp, as described in Chapter 2. For example:

```
CL-USER> (ql:quickload "cffi")
```

The examples below assume you have loaded the library named in each section. The commercial implementations also have interfaces of their own for most of these things, described in their manuals.

4.1 The Operating System

UIOP is a library of portable utilities for dealing with the operating system. It is part of ASDF, so it is already there. Its `run-program` runs another program and can return what it printed:

```
CL-USER> (uiop:run-program '("date" "+%Y")
                        :output '(:string :stripped t))
"2026"
NIL
0
```

The three values are the program's output, its error output, which was not collected here, and its exit code. Giving the command as a list of strings, as above, passes the arguments straight to the program. You can also give one string, which is handed to the shell.

Environment variables are read with `uiop:getenv`, which returns `nil` if the variable is not set:

```
CL-USER> (uiop:getenv "HOME")
"/home/reader"
```

UIOP also has functions for the file system, such as `directory-files`, `copy-file`, `ensure-directories-exist` and `read-file-string`. Where a Lisp function will do the job, prefer it to running a shell command: it works the same on every operating system, and when it fails you get a Lisp condition you can handle.

4.2 Foreign Functions

Most operating systems and a great many libraries present their interface as C functions. CFFI, the Common Foreign Function Interface (<https://cffi.common-lisp.dev>), lets Lisp call them. The steps are:

1. Load the shared library that contains the functions (a `.so` file on Linux, `.dylib` on macOS, `.dll` on Windows). The C standard library is already loaded.
2. Tell Lisp the name of each function and the types of its arguments and result.
3. Call it.

For a single call, `foreign-funcall` does the last two steps at once. Here is the C function `strlen`:

```
CL-USER> (cffi:foreign-funcall "strlen" :string "Hello, C!" :size)
9
```

The arguments come in pairs of a type and a value, and the type of the result comes last. To make a C function callable like a Lisp function, define it once with `defcfun`:

```
CL-USER> (cffi:defcfun ("getpid" process-id) :int)
PROCESS-ID
CL-USER> (process-id)
4242
```

A great many C libraries have already been wrapped this way and are in Quicklisp, so look there before you write your own.

4.3 Threads

Every current implementation can run several threads at once. Each has its own functions for this, and the Bordeaux Threads library puts one interface over all of them. Its current interface is in the package `bt2`:

```
(defparameter *counter* 0)
(defparameter *lock* (bt2:make-lock))

(defun count-up (times)
  (dotimes (n times)
    (bt2:with-lock-held (*lock*)
      (incf *counter*)))))
```

```
CL-USER> (let ((threads (loop repeat 4
                                collect (bt2:make-thread
                                          (lambda () (count-up 100000))))))
          (mapc #'bt2:join-thread threads)
          *counter*)
400000
```

Four threads each add to the counter a hundred thousand times. `Join-thread` waits for a thread to finish. The lock makes sure that only one thread changes the counter at a time; without it, some of the additions would be lost. All the threads share the same memory, so data that more than one of them changes needs a lock.

A dynamic binding made with `let` in one thread, on the other hand, is seen only by that thread, as Chapter 3 mentioned.

4.4 TCP/IP Sockets

The `usocket` library gives portable access to network sockets. A connected socket gives you a stream, and you read and write it with the same functions as a file.

Here is a server that accepts one connection, reads a line from it, sends an answer, and closes:

```
(defun serve-once (port)
  (let ((listener (usocket:socket-listen "127.0.0.1" port
                                          :reuse-address t)))
    (unwind-protect
      (let* ((connection (usocket:socket-accept listener))
             (stream (usocket:socket-stream connection)))
        (unwind-protect
          (let ((command (read-line stream)))
            (format stream "You said: ~a~%" command)
            (finish-output stream))
          (usocket:socket-close connection)))
      (usocket:socket-close listener))))
```

and a client that connects, sends a line, and returns the reply:

```
(defun ask (port message)
  (let* ((socket (usocket:socket-connect "127.0.0.1" port))
         (stream (usocket:socket-stream socket)))
    (unwind-protect
      (progn
        (write-line message stream)
        (finish-output stream)
        (read-line stream))
      (usocket:socket-close socket))))
```

To try them in one Lisp, run the server in a thread of its own, since it waits for a connection:

```
CL-USER> (bt2:make-thread (lambda () (serve-once 4444)))
#<BT2:THREAD "Unknown thread 4" {1206502383}>
CL-USER> (ask 4444 "hello")
"You said: hello"
NIL
```

They would work just as well in two Lisps on two machines, with the server's address in place of 127.0.0.1. The calls to `finish-output` matter: output to a socket is buffered, and nothing is sent until the buffer fills or you ask. The `unwind-protect` forms make sure that the sockets are closed whatever happens.

A real server would handle many connections and deal with errors. For the usual protocols you will not write that yourself. The next sections use libraries built on top of sockets.

4.5 JSON

Much of the data that programs exchange is JSON. Several libraries read and write it. One good current choice is `jzon`, loaded as `com.inuoe.jzon`. A package name that long is a case for the local nicknames of Section 3.9; here we write it out.

```
CL-USER> (defparameter *city*
           (com.inuoe.jzon:parse
            "{ \"name\": \"Detroit\", \"tags\": [\"motor\", \"music\"] }"))
*CITY*
CL-USER> (gethash "name" *city*)
"Detroit"
T
CL-USER> (gethash "tags" *city*)
#("motor" "music")
T
```

A JSON object becomes a hash table with string keys, and a JSON array becomes a vector. `Stringify` goes the other way:

```
CL-USER> (com.inuoe.jzon:stringify (list 1 2 "three"))
"[1,2,\"three\"]"
```

4.6 Databases

There are libraries for all the common database systems, among them `Postmodern` for PostgreSQL, `cl-mysql`, and `cl-dbi`, which gives one interface to several. The simplest to try is SQLite, which keeps a whole database in one file and needs no server. The Lisp library is called `sqlite`, and it uses the SQLite C library, which must be installed on your machine.

```
CL-USER> (defparameter *db* (sqlite:connect ":memory:"))
*DB*
```

```
CL-USER> (sqlite:execute-non-query *db*
          "create table capitals (state text, city text)")
NIL
CL-USER> (sqlite:execute-non-query *db*
          "insert into capitals values (?, ?)" "Michigan" "Lansing")
NIL
CL-USER> (sqlite:execute-non-query *db*
          "insert into capitals values (?, ?)" "Illinois" "Springfield")
NIL
CL-USER> (sqlite:execute-to-list *db*
          "select * from capitals order by state")
(("Illinois" "Springfield") ("Michigan" "Lansing"))
CL-USER> (sqlite:execute-single *db*
          "select city from capitals where state = ?" "Michigan")
"Lansing"
CL-USER> (sqlite:disconnect *db*)
```

The name `":memory:"` asks for a temporary database in memory; give a file name to keep one on disk. Each row comes back as a list. Always pass values to a query through the question-mark placeholders, as here, and never by pasting them into the SQL text. That is what keeps hostile input from being run as SQL.

4.7 Web Serving

A web server written in Lisp runs inside your Lisp image. That has a pleasant consequence: all of the program's state is right there in memory between requests, and you can redefine the function behind a page while the server is running and see the change on the next request.

Hunchentoot (<https://edic1.github.io/hunchentoot/>) is a widely used web server library. Start a server on a port:

```
CL-USER> (defparameter *server*
          (hunchentoot:start
           (make-instance 'hunchentoot:easy-acceptor :port 8000)))
*SERVER*
```

and define a page:

```
(hunchentoot:define-easy-handler (hello :uri "/hello")
  ((name :init-form "World"))
  (setf (hunchentoot:content-type*) "text/plain")
  (format nil "Hello, ~a!" name))
```

Now visit <http://localhost:8000/hello> in your browser and you will see “Hello, World!” The handler is a function. Whatever it returns is sent to the browser. The parameter `name` is filled in from the request, so <http://localhost:8000/hello?name=Lisp> answers “Hello, Lisp!” The same parameter is filled in from a posted form.

Try it: With the server running, change the greeting in the handler, evaluate the definition again, and reload the page.

You can make requests from Lisp as well. Dexador and Drakma are HTTP client libraries:

```
CL-USER> (dexador:get "http://localhost:8000/hello?name=Lisp")
"Hello, Lisp!"
200
...
```

The first two values are the body of the response and the HTTP status code. To stop the server, call `(hunchentoot:stop *server*)`.

To produce HTML you can use `format`, or better, one of the libraries that write HTML from Lisp lists, such as Spinneret or CL-WHO. For larger applications there are frameworks built on these pieces. Hunchentoot is not the only server: Franz's AllegroServe, used in *Basic Lisp Techniques*, this book's predecessor, is open source and runs on several implementations, and Clack and Woo are other popular choices.

4.8 Embedded Languages

Chapter 3 showed that macros let you add operators to Common Lisp. Carried further, a set of macros and functions becomes a language in its own right for some kind of problem, sitting on top of CL. This is called an *embedded language*, or a domain-specific language. The `loop` macro and the HTML libraries just mentioned are small ones.

A larger example is Gendl (<https://gendl.org>), an open-source language for engineering and design applications. Its main operator, `define-object`, is to Gendl what `defclass` is to CLOS. The objects it defines compute their values on demand, remember them, and keep track of what depends on what, so that when an input changes, whatever was computed from it is worked out again the next time it is asked for. A Gendl program is still a Common Lisp program. It is compiled by the same compiler, loaded with ASDF, and developed at the same REPL, with all of CL available inside it.

That is the general pattern. Because the embedded language is made of ordinary macros, it needs no parser, no separate compiler and no bridge to the host language, and its users keep the whole of Common Lisp. Building a language up toward the problem, until the program says what it means in the problem's own terms, is the style of programming that Lisp was designed for.

Chapter 5

Where to Go Next

This book has covered the ground you need in order to start. Here is where to go for the rest.

5.1 Reference

- **The Common Lisp HyperSpec**, <https://www.lispworks.com/documentation/HyperSpec/Front/index.htm>. The text of the standard as a hypertext document. It is the final word on what any standard function does. It is written for implementers, and its entries are worth learning to read. Your editor can open the entry for the name at the cursor.
- **The Common Lisp Cookbook**, at <https://lispcookbook.github.io/cl-cookbook/>. Task-oriented chapters on strings, files, dates, regular expressions, testing, web development, and much else, kept up to date by the community. This is the best next stop after this book.
- **The manual for your implementation**. Each one documents its extensions, its compiler, its debugger and its profiler. SBCL's is at <https://www.sbcl.org/manual/>.
- **The ASDF manual**, <https://asdf.common-lisp.dev>, for everything about defining and building systems.

5.2 Books

- **Practical Common Lisp**, Peter Seibel, Apress, 2005. A full introduction for working programmers, with chapters that build real programs. Free to read at <https://gigamonkeys.com/book/>.
- **Common Lisp: A Gentle Introduction to Symbolic Computation**, David S. Touretzky. Starts from nothing and goes slowly. A good choice if this book moved too fast. Free at <https://www.cs.cmu.edu/~dst/LispBook/>.

- **ANSI Common Lisp**, Paul Graham, Prentice Hall, 1996. A compact tutorial followed by a compact reference.
- **On Lisp**, Paul Graham, Prentice Hall, 1993. The classic book about macros and building languages on Lisp. Free at <https://www.paulgraham.com/onlisp.html>.
- **Paradigms of Artificial Intelligence Programming**, Peter Norvig, Morgan Kaufmann, 1992. Teaches programming at a high level through substantial programs in Common Lisp. Free at <https://github.com/norvig/paip-lisp>.
- **Object-Oriented Programming in Common Lisp**, Sonya Keene, Addison-Wesley, 1989. The standard introduction to CLOS.
- **Common Lisp Recipes**, Edmund Weitz, Apress, 2016. Hundreds of specific problems and their solutions, with explanations of why.

5.3 Libraries and Examples

To find a library for a job, start with the Awesome CL list, <https://github.com/CodyReichert/awesome-cl>, which sorts the well-regarded libraries by subject, or search Quicklisp from the REPL with `ql:system-apropos`. CLiki, <https://www.cliki.net>, is the community's long-running wiki.

The best way to get better at Lisp is to read programs. Every library you load with Quicklisp is on your disk in source form, and `M-` in your editor takes you from any function's name to its definition. Read the code of the libraries you use. Much of it is short and very good.

For complete applications to read and run, see the sites above, and for engineering and web applications built with Gendl, the examples at <https://github.com/genworks/demos>.

5.4 Community

<https://common-lisp.net> hosts many of the central projects and has pointers to the rest. Planet Lisp, <https://planet.lisp.org>, collects the blogs of people who write about Lisp. Questions are welcome in the `r/Common_Lisp` forum on Reddit, on the Lisp Discord server, and in the `#commonlisp` channel on the Libera.Chat IRC network. The European Lisp Symposium meets every spring, and its papers and talks are online.

5.5 A Last Word

The way of working this book has described, with a running image, a REPL, and a program built and changed a piece at a time, will feel odd for a few days if you are used to compiling and running whole programs. Stay with it. Start small, try each piece as you write it, ask the running Lisp when you are not sure, and let the language grow toward your problem.

Index

ABCL, 5
AI agents, 14
Alive, 9
Allegro CL, 5
and, 29
ANSI standard, 1
append, 26
apply, 31
apropos, 13
aref, 37
arrays, 37
ASDF, 10

backquote, 39
Bordeaux Threads, 48
break, 13

car, 24
case, 30
CCL, 5
cdr, 24
CFFI, 48
classes, 38
CLOS, 38
closures, 32
comments, 11
Common Lisp, 1
compile-file, 7
cond, 29
conditions, 40
cons, 26
cons cells, 26
consing, 26
Cookbook, 53

databases, 50
debugger, 12
defcfun, 48
defclass, 38
defmacro, 39
defpackage, 41
defparameter, 21
defstruct, 37
defun, 21
defvar, 21
describe, 13
destructive operations, 27
Docker, 6
documentation, 13
documentation strings, 21
dolist, 30
dotimes, 30
dynamic scope, 23
dynamic typing, 2, 19

ecase, 30
ECL, 5
Emacs, 8
embedded languages, 52
environment variables, 47
eql, 44
equal, 44
equality, 44
error, 40
evaluation, 17

fasl file, 7
first, 23
format, 33
funcall, 31
functions
 as objects, 31

garbage collection, 2
Gendl, 52
generic functions, 38

- `getf`, 28
- `gethash`, 36
- global variables, 21
- `handler-case`, 40
- hash tables, 36
- higher-order functions, 31
- Hunchentoot, 51
- HyperSpec, 53
- `if`, 25
- image, 3, 11
- implementations, 5
- init file, 9
- iteration, 30
- JSON, 50
- keyword arguments, 32
- keywords, 20, 42
- `lambda`, 31
- Lem, 9
- length, 25
- `let`, 22
- `let*`, 22
- lexical scope, 23
- LispWorks, 5
- `list`, 21
- lists, 20
- load, 7
- local variables, 22
- loop, 30
- `macroexpand-1`, 39
- macros, 18, 39
- `mapcar`, 28
- McCarthy, John, 1
- `member`, 25
- methods, 38
- Model Context Protocol, 14
- multiple values, 33
- `multiple-value-bind`, 33
- `nil`, 24
- `not`, 29
- `null`, 24
- numbers, 19
- optional arguments, 32
- `or`, 29
- package-local nicknames, 42
- packages, 41
- pathnames, 35
- `pop`, 26
- predicates, 24
- prefix notation, 17
- `princ`, 34
- `print`, 34
- `probe-file`, 35
- `progn`, 29
- property lists, 28
- `push`, 26
- Quicklisp, 9
- `quote`, 19
- ratios, 20
- `read`, 35
- `read-line`, 35
- recursion, 25
- `reduce`, 28
- `remove`, 27
- REPL, 3, 6
- `rest`, 24
- rest arguments, 33
- restarts, 12, 41
- `run-program`, 47
- SBCL, 5
- scripting, 11
- sequences, 26
- `setf`, 22, 36
- `setq`, 22
- SLIME, 8
- Sly, 8
- sockets, 49
- `sort`, 27
- special operators, 18
- special variables, 23
- streams, 33
- strings, 20
- structures, 37

- subseq, 26
- symbols, 20
- system, 10

- t, 24
- threads, 48
- time, 14
- trace, 13

- UIOP, 47
- unless, 29
- unwind-protect, 41
- usocket, 49

- vectors, 37

- web serving, 51
- when, 29
- with-open-file, 35