Basic Lisp Techniques

David J. Cooper, Jr.

March 19, 2003

Foreword[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.

[1] this Foreword was authored by Franz Inc.

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

Chapter 1

Introduction

1.1 The Past, Present, and Future of Common Lisp

1.1.1 Lisp Yesterday

John McCarthy discovered the basic principles of Lisp in 1958, when he was processing complex mathematical lists at MIT. Common Lisp (CL) is a high-level computer language, whose syntax follows a simple list-like structure. The term "Lisp" itself originally stood for "LISt Processing." When developing, testing, and running a CL program, at the core is a modern-day version of the original List Processor which processes (compiles, evaluates, etc.) the elements of your program. These elements, at the source code level, are represented as lists. A *list*, in this context, is just a sequence of items, much like a familiar shopping list or checklist. Originally the list was pretty much the only data structure supported by Lisp, but modern-day Common Lisp supports a wide range of flexible and efficient data structures.

A typical Common Lisp development and runtime environment behaves like a complete operating system, with multiple threads of execution and the ability to load new code and redefine objects (functions, etc.) *dynamically*, i.e. without stopping and restarting the "machine."

This "operating system" characteristic also makes CL significantly more flexible than other popular programming languages. Whereas some other languages, such as shell scripts or Perl CGI scripts, need to pipe data around, in CL all data manipulations can run interactively within one single process with a shared memory space.

1.1.2 Lisp Today

The most popular form of Lisp used today, "ANSI Common Lisp," was initially designed by Guy Steele in Common Lisp, the Language, 2nd Edition ("CLtL2") — (see the Bibliography in Appendix B). This version of Lisp became the accepted industry standard. In 1995, the American National Standards Institute recognized a slightly updated version as ANSI Common Lisp, the *first* object-oriented language to receive certification, and ANSI CL remains the *only* language that meets all of the criteria set forth by the Object Management Group (OMG) for a complete object-oriented language.

Paul Graham outlines ANSI CL in his 1996 book ANSI Common Lisp, also listed in Appendix B.

Official language standardization is important, because it protects developers from becoming saddled with legacy applications as new versions of a language are implemented. For example, this lack of standardization has been a continuing problem for Perl developers. Since each implementation of Perl defines the behavior of the language, it is not uncommon to have applications that require outdated versions of Perl, making it nearly impossible to use in a mission-critical commercial environment.

1.1.3 Lisp Tomorrow

Many developers once hoped that the software development process of the future would be more automated through Computer-aided Software Engineering (CASE) tools. Such tools claim to enable programmers and non-programmers to diagram their applications visually and automatically generate code. While useful to a certain extent, traditional CASE tools cannot cover domain-specific details and support all possible kinds of customizations — so developers inevitably need to hand-code the rest of their applications, breaking the link between the CASE model and the code. CASE systems fall short of being a true "problem-solving tool."

The recursive nature of CL, and its natural ability to build applications in layers and build upon itself — in essence, *code which writes code which writes code...*, makes CL a much better software engineering solution. CL programs can generate other CL programs (usually at compile-time), allowing the developer to define *all* parts of the application in one unified high-level language. Using macros and functions, naturally supported by the CL syntax, the system can convert applications written in the high-level language automatically into "final" code. Thus, CL is both a programming language and a powerful CASE tool, and there is no need to break the connection.

Several other features of CL also save significant development time and manpower:

- Automatic Memory Management/Garbage Collection: inherent in CL's basic architecture.
- Dynamic Typing: In CL, values have types, but variables do not. This means that you don't have to declare variables, or placeholders, ahead of time to be of a particular type; you can simply create them or modify them on the fly[1].
- Dynamic Redefinition: You can add new operations and functionality to the running CL process without the need for downtime. In fact, a program can be running in one thread, while parts of the code are redefined in another thread. Moreover, as soon as the redefinitions are finished, the application will use the new code and is effectively modified while running. This feature is especially useful for server applications that cannot afford downtimes – you can patch and repair them while they are running.
- Portability: The "abstract machine" quality of CL makes programs especially portable.

[1] While you don't *have* to declare variable types, you *may* declare them. Doing so can help CL's compiler to optimize your program further.

- Standard Features: CL contains many built-in features and data structures that are non-standard in other languages. Features such as full-blown symbol tables and package systems, hash tables, list structures, and a comprehensive object system are several "automatic" features which would require significant developer time to recreate in other languages.

1.2 Convergence of Hardware and Software

The most common criticism of Lisp usually made by programmers unfamiliar with the language is that Lisp applications are too big and too slow. While this may have been true 20 years ago, it is *absolutely* not the case today. Today's inexpensive commodity computers have more memory than some of the most powerful machines available just five years ago, and easily meet the computing needs of a typical CL application. Hardware is no longer the critical factor. Programmers and development time are!

Further, CL programming teams tend to be small, because the inherent features in CL empower developers to do more. Without excessive effort, a single CL developer can create and deploy a sophisticated, powerful program in a much shorter period of time than if another language were used.

1.3 The CL Model of Computation

Programming in CL is distinguished from programming in other languages due to its unique syntax and development mode. CL allows developers to make changes and test them *immediately*, in an incremental manner. In other languages, a developer must follow the "compile-link-run-test" cycle. These extra steps add minutes or hours to *each cycle* of development, and break the programmer's thought process. Multiply these figures by days, weeks, and months — and the potential savings are phenomenal.

Chapter 2

Operating a CL Development Environment

This chapter covers some of the hands-on techniques used when working with a CL development environment, specifically, the Allegro CL ® environment from Franz Inc. If you are completely unfamiliar with any Lisp language, you may wish to reference Chapter 3 as you read this chapter.

Ideally, while reading this chapter, you should have access to an actual Common Lisp session and try typing in some of the examples.

When working with Common Lisp, it helps to think of the environment as its own operating system, sitting on top of whatever operating system you happen to be working with. In fact, people used to build entire *workstations* and *operating systems* in Lisp. The Symbolics Common Lisp environment still runs as an emulated machine on top of the 64-bit Compaq/DEC Alpha CPU, with its own Lisp-based operating system.

A complete and current listing of available CL systems can be found on the Association of Lisp Users website, at

`http://www.alu.org/table/systems.htm`

These systems follow the same basic principles, and most of what you learn and write on one system can be applied and ported over to others. The examples in this guide were prepared using Allegro CL for Linux. Where possible, we will note Allegro CL-specific syntax and extensions.

2.1 Installing a CL Environment

Installing a CL environment is generally a straightforward process. For Allegro CL on Linux, this basically involves running a simple shell script which steps you through some licensing information, then proceeds to unpack the distribution, and sets up the CL executable and supporting files.

The Windows installation involves somewhat less typing and more mouse clicking.

2.2 Running CL in a Shell Window

The most primitive and simplest way to run CL is directly from the Unix (Linux) shell from within a terminal window. Most programmers do not work this way on a daily basis because other modes of use provide more power and convenience with fewer keystrokes.

This mode of working is sometimes useful for logging into a running CL system on a remote host (e.g. a CL process acting as a webserver) when using a slow dial-up connection.

2.2.1 Starting CL from a Terminal Window

To start a CL session from a terminal window, simply type

```
alisp
```

from the Unix (or DOS) command line. Assuming your shell execution path is set up properly and CL is installed properly, you will see some introductory information, then will be presented with a Command Prompt which should look very similar to the following:

```
CL-USER(1):
```

CL has entered its *read-eval-print* loop, and is waiting for you to type something which it will then *read*, *evaluate* to obtain a *return-value*, and finally *print* this resulting return-value.

The `USER` printed before the prompt refers to the current CL *package* (more on Packages later), and the number (`1`) simply keeps track of how many commands have been entered at the prompt.

2.2.2 Stopping CL from a Terminal Window

Now that you have learned how to start CL, you should probably know how to stop it. In Allegro CL, one easy way to stop the CL process is to type

```
(excl:exit)
```

at the Command Prompt. This should return you to the shell. In very rare cases, a stronger hammer is required to stop the CL process, in which case you can try typing

```
(excl:exit 0 :no-unwind t)
```

Try It: Now try starting and stopping CL several times, to get a feel for it. In general, you should notice that CL starts much more quickly on subsequent invocations. This is because the entire executable file has already been read into the computer's memory and does not have to be read from the disk every time.

A shortcut for the `(excl:exit)` command is the *toplevel* command `:exit`. Toplevel commands are words which are preceded by a colon [`:`] that you can type at the CL Command Prompt as a shorthand way of making CL do something.

Try It: Start the CL environment. Then, using your favorite text editor, create a file `/tmp/hello.lisp` and place the following text into it (don't worry about understanding the text for now):

```
(in-package :user)

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

Save the file to disk. Now, at the CL prompt in the shell where you started the CL environment, compile and load this file as follows *(text after the "USER" prompt represents what you have to type; everything else is text printed by CL):*

```
CL-USER(3): (compile-file "/tmp/hello")
;;; Compiling file /tmp/hello.lisp
;;; Writing fasl file /tmp/hello.fasl Fasl write complete
#p"/tmp/hello.fasl"
NIL
NIL

CL-USER(4): (load "/tmp/hello")
; Fast loading /tmp/hello.fasl
T
```

By default, the `compile-file` command will look for files with the `.lisp` suffix, and the `load` command will load the resulting compiled machine-code (binary) file, which by default will have the extension `.fasl`. The extension "`.fasl`" stands for "FASt Loading" file.

Note that, in general, simply loading an uncompiled Lisp file (using `load`) will functionally have the same effect as loading a compiled binary (`fasl`) file. But the `fasl` file will load faster, and any functions, objects, etc. defined in it will perform faster, since they will have been optimized and translated into language which is closer to the actual machine. Unlike Java "`class`" files, CL `fasl` files usually represent native machine-specific code. This means that compiled CL programs will generally run much faster than compiled Java programs, but CL programs must be compiled separately for each type of machine on which you want to run them.

Finally, try executing the newly defined function `hello` by typing the following at your command line:

```
CL-USER(5): (hello)
Hello, World!
"Hello, World!"
```

You should see the String `Hello, World!` printed (without double-quotes), then the returned String (with double-quotes) will be printed as well. In the next chapter, we will learn more about exactly what is going on here. For now the main point to understand is the idea of compiling and loading definitions from files, and invoking functions by typing expressions at the CL command line.

2.3 Running CL inside a Text Editor

A more powerful way of developing with CL is by running it from within a Unix (or DOS) shell that is running inside a buffer in a powerful text editor, such as Gnu Emacs[1]. One advantage of this approach is that the editor (in our example, Emacs) keeps a history of the entire session: both the expressions the programmer has typed, as well as everything that CL has printed. All these items can be easily reviewed or recalled.

2.3.1 A Note on Emacs and Text Editors

To develop using Common Lisp, you can, of course, use any text editor of your choice, although Emacs happens to be especially synergistic for working with CL, for reasons we will discuss later. With a bit of practice, you will find that the combination of Emacs and Allegro CL provides a more powerful and convenient development environment than any of today's flashy "visual programming" environments. If you are used to working with the "vi" editor, Emacs comes with several "vi" emulation modes which you may want to experiment with on a transitional basis. If you are used to working with a standard Windows-based text editor, you will find a smooth transition to Emacs.

This guide provides examples using Gnu Emacs.

To work with CL in an Emacs shell buffer, you should be reasonably familiar with the text editor. If you are completely new to Gnu Emacs, go through the Emacs Tutorial, available under the "Help" menu at the top of the Emacs window.

2.3.2 Emacs Terminology

As you will learn in the Emacs Tutorial, when working with Emacs you make frequent use of *key chords*. Similar to "chords" on a piano, keychords can allow powerful text processing with relatively few sequential keystrokes.

They are key combinations involving holding down the `Control` key and/or the `Meta` key, and pressing some other regular key.

See Appendix C for information on which keys represent `Control` and `Meta` on your keyboard, and how you can customize these. To begin, try using the key labeled "Control" for `Control` and the keys labeled "Alt" (or the diamond-shaped key on Sun keyboards) for `Meta`. If your keyboard has a "Caps Lock" key to the left of the letter "A," you should consider remapping this to function as `Control`, as explained in Appendix C.

This guide uses the following notations to indicate keychords (identical to those referenced in the Emacs Tutorial):

- `M-x` means to hold down the `Meta` key, and press (in this case) the "X" key
- `C-x` means to hold down the `Control` key, and press (in this case) the "X" key
- `C-M-q` means to hold down the `Control` key *and* the `Meta` key at the *same time* , and press (in this case) the "Q" key

[1]Gnu Emacs comes pre-installed with all Linux systems, and for other systems should come as part of your Allegro CL distribution. In any case, Emacs distributions and documentation are available from `http://www.gnu.org`

- `M-A` would mean to hold down the `Meta` key, *and* the `Shift` key (because the "A" is uppercase), and press (in this case) the "A" key.

2.3.3 Starting, Stopping, and Working With CL inside an Emacs Shell

To start Emacs, type

```
emacs
```

or

```
gnuemacs
```

then start a Unix (or DOS) shell within emacs by typing `M-x shell`. Now type

```
lisp
```

inside this shell's window to start a CL session there. To shut down a session, exit CL exactly as above by typing

```
(excl:exit)
```

Exit from the shell by typing

```
exit
```

and finally exit from Emacs by typing `C-x C-c` or `M-x kill-emacs`. Note that it is good practice always to exit from CL before exiting from Emacs. Otherwise the CL process may be left as an "undead" (zombie) process.

When in the CL process, you can move forward and backward in the "stack" (history) of expressions that you have typed, effectively recalling previous commands with only a few keystrokes. Use `M-p` to move backward in the history, and `M-n` to move forward. You can also use all of the editor's text processing commands, for example to cut, copy, and paste expressions between the command line and other windows and open files ("buffers") you may have open.

Try It: try typing the following expressions at the CL prompt. See what return-values are printed, and try recalling previous expressions (commands) using `M-p` and `M-n`:

```
(list 1 2 3)

(+ 1 2 3)

(> 3 4)

(< 3 4)
```

Now, of course, you can edit, compile, load, and test your `hello.lisp` file as we did in Section 2.2. But this time you are staying within a single environment (the text editor environment) to achieve all these tasks.

2.4 Running CL as a subprocess of Emacs

An even more powerful way of developing with Allegro CL is by running it within Emacs in conjunction with a special Emacs/Lisp interface provided by Franz Inc. This technique provides all the benefits of running in an Emacs shell as outlined above, but is more powerful because the Emacs-CL interface extends Emacs with several special commands. These commands interact with the CL process, making the editor appear to be "Lisp-aware." The combination of Emacs with the Emacs-CL interface results in a development environment whose utility approaches that of the vintage Symbolics Lisp Machines, which many Lisp aficionados still consider to be technology advanced well beyond anything produced today.

Complete documentation of the Emacs-CL interface can be viewed by pointing your web browser at: `file:/usr/local/acl62/doc/eli.htm` (you may have to modify this pathname to point to your actual installed location of Allegro CL (`acl62`)).

2.4.1 Starting the CL subprocess within Emacs

To enable your Emacs to function in the proper manner, place the following expressions in a file named `.emacs` in your home directory:

```
(setq load-path
  (cons "/usr/local/acl62/eli" load-path))
(load "fi-site-init")
```

As above, you may have to modify this pathname to point to your actual installed location of Allegro CL (`acl62`). Once you have added these lines to your `.emacs` file, you should be able to restart emacs, then issue the command:

```
M-x fi:common-lisp
```

to start CL running and automatically establish a network connection between the Emacs process and the CL process (in theory, the two processes could be running on different machines, but in practice they usually will run on the same machine). Accept all the defaults when prompted.

You can also start emacs with the CL subprocess automatically from a terminal window, by invoking emacs with an optional "function" argument, as follows:

```
emacs -f fi:common-lisp
```

This will start Emacs, which itself will immediately launch CL.

2.4.2 Working with CL as an Emacs subprocess

Once you have started CL within Emacs, you will be presented with a special buffer named `*common-lisp*` which in many ways is similar to simply running CL within a shell within Emacs as above. But now you have several more capabilities available to you, as outlined in Table 2.1.

To summarize, the Lisp-Emacs interface adds capabilities such as the following to your development environment:

Keychord	*Action*	*Emacs-Lisp function*
`C-M-x`	Compiles and loads the function near the Emacs point	`fi:lisp-eval-or-compile-defun`
`C-c C-b`	Compiles and loads the current Emacs buffer	`fi:lisp-eval-or-compile-current-buffer`
`M-A`	Shows the Argument List for a CL function	`fi:lisp-arglist`
`C-c .`	Finds the definition of a CL object	`fi:lisp-find-definition`

Table 2.1: Common Commands of the Emacs-Lisp Interface

- Start and stop the CL process using Emacs commands
- Compile and load files into the running CL process, directly from the open Emacs buffer you are editing (no need explicitly to save the file, call the `compile-file` function, then call the `load` function, as with the "in-a-shell" techniques above)
- Query the running CL process for information about objects currently defined in it — for example, querying the CL, directly from Emacs, for the argument List of a particular function
- Run multiple *read-eval-print* loops, or *listeners*, in different threads of CL execution, as Emacs buffers
- Compile and load specified sections of an open Emacs buffer
- Locate the source code file corresponding to any defined CL object, such as a function or parameter, and automatically open it in an Emacs buffer
- Perform various debugging actions and other general CL help capabilities, directly from within the Emacs session

In short, Emacs becomes a remarkably powerful, infinitely customizable, CL integrated development environment.

2.4.3 Compiling and Loading a File from an Emacs buffer

Try it: Try starting Emacs with a CL subprocess as outlined above. If you have trouble, consult the URL listed above for complete instructions on running the Lisp-Emacs interface.

Once you have Emacs running with a CL subprocess, you should have a buffer named `*common-lisp*`. If your Emacs session is not already visiting this buffer, visit it with the command `C-x b *common-lisp*`, or select `*common-lisp*` from the "Buffers" menu in Emacs (see Appendix C for instructions on how to set up a "hot key" for this and other common tasks when working with the Emacs-Lisp interface).

Now, split your Emacs frame into two windows, with the command `C-x 2`. In both windows you should now see `*common-lisp*`. Now move the point[2] to the other window with `C-x o`, and open the `hello.lisp` file you created for the previous exercise[3]. If you don't have that file handy, create it now, with the following contents:

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

Now, compile and load the contents of this function definition with `C-M-x`. Finally, switch to the `*common-lisp*` window again with `C-x o`, and try invoking the `hello` function by calling it in normal Lisp fashion:

```
CL-USER(5): (hello)
Hello, World!
"Hello, World!"
```

You should see the results printed into the same `*common-lisp*` buffer.

Try it: Now try the following: go back to the `hello.lisp` buffer, make an edit to the contents (for example, change the String to `"Hello, CL World!"`). Now compile the buffer with this change (Note that you do not necessarily have to save the file). Finally, return to the `*common-lisp*` buffer, and try invoking the function again to confirm that the redefinition has taken effect.

You have now learned the basic essentials for working with the Emacs-Lisp interface. See Appendix C for more information about Emacs customization and convenient Emacs commands specific to the Lisp editing mode.

2.5 Integrated Development Environment

Some CL implementations support or integrate with an *IDE*, or *Integrated Development Environment*. Such environments provide visual layout and design capabilities for User Interfaces and other graphical, visual techniques for application development. For example, such a system is available for Allegro CL on the Microsoft platforms. The details of these systems are covered in their own documentation and are beyond the scope of this guide.

2.6 The User Init File

When CL begins execution, it usually looks for an *initialization file*, or "init" file, to load. When CL loads this file, it will read and evaluate all the commands contained in it.

The default name and location for this initialization file is

`.clinit.cl`

[2] In technical Emacs parlance, the *point* is the insertion point for text, and the *cursor* is the mouse pointer.

[3] Remember that pressing the space bar will automatically complete filenames for you when opening files in Emacs.

in the user's home directory on Unix/Linux, and

```
c:\clinit.cl
```

on Windows systems.

For example, the init file can be used to define library functions and variables that the user employs frequently, or to control other aspects of the CL environment. An example init file is:

```
(in-package :user)

(defun load-project ()
  (load "~/lisp/part1")
  (load "~/lisp/part2"))
```

This will have the effect of defining a convenient function the user can then invoke to load the files for the current project. Note the call to the `in-package` function, which tells CL which Package to make "current" when loading this file. As we will see in more detail in Section 3.7, CL Packages allow programs to be separated into different "layers," or "modules." Notice also that in the calls to `load` we do not specify a file "type" extension, e.g. `.lisp` or `.fasl`. We are making use of the fact that the `load` function will look first for a binary file (of type `fasl`), and if this is not found it will then look for a file of type `cl` or `lisp`. In this example we assume that our Lisp files will already have been compiled into up-to-date binary `fasl` files.

In Section 2.9 we will look at some techniques for managing larger projects, and automating both the compiling and loading process for the source files making up a project.

Note: the IDE mentioned in Section 2.5 is also a full-featured and powerful project system. The IDE is described in its own documentation, and is beyond the scope of this document.

2.7 Using CL as a scripting language

Although CL is usually used in the interactive mode described above while developing a program, it can be also used as a scripting language in more of a "batch" mode. For example, by running CL as:

```
lisp -e '(load "~/script")' < datafile1 > datafile2
```

it will load all of the function definitions and other expressions in file `script.fasl` or `script.lisp`, will process `datafile1` on its Standard Input, and will create `datafile2` with anything it writes to its Standard Output. Here is an example of a simple `script.lisp` file:

```
(in-package :user)

(defun process-data ()
```

```
(let (x j)
  ;;Read a line of input from datafile1
  (setq x (read-line))
  ;;Remove leading spaces from X
  (setq j (dotimes (i (length x))
            (when (not (string-equal " " (subseq x i (+ i 1))))
              (return i))))
  (setq x (subseq x j))
  ;;Write the result to datafile2
  (format t "~A~%" x)))

(process-data)
```

This file defines a function to process the datafile, then calls this function to cause the actual processing to occur.

Note the use of the semicolon (";"), which is a *reader macro* which tells CL to treat everything between it and the end of the line as a comment (to be ignored by the CL Reader).

2.8 Debugging in CL

CL provides one of the most advanced debugging and error-handling capabilities of any language or environment. The most obvious consequence of this is that your application will *very* rarely "crash" in a fatal manner. When an error occurs in a CL program, a *break* occurs. CL prints an error message, enters the debugger, and presents the user with one or more possible restart actions. This is similar to running inside a debugger, or in "debug mode," in other languages, but in CL this ability comes as a built-in part of the language.

For example, if we call the + function with an alphabetic Symbol instead of a Number, we generate the following error:

```
CL-USER(95): (+ 'r 4)
Error: `R' is not of the expected type `NUMBER'
   [condition type: TYPE-ERROR]

Restart actions (select using :continue):
  0: Return to Top Level (an "abort" restart)
[1] CL-USER(96):
```

The number [1] at the front of the prompt indicates that we are in the debugger, at Level 1. The debugger is itself a CL Command Prompt just like the toplevel Command Prompt, but it has some additional functionality. The next section provides a partial list of commands available to be entered directly at the debugger's Command Prompt:

2.8.1 Common debugger commands

- *:reset* returns to the toplevel (out of any debug levels)

- *:pop* returns up one level
- *:continue* continues execution after a break or an error
- *:zoom* Views the current function call stack[4]
- *:up* Moves the currency pointer up one frame in the stack
- *:down* Moves the currency pointer down one frame in the stack
- *:help* Describes all debugger commands

Franz Inc.'s Allegro Composer product also offers graphical access to these commands, as well as additional functionality.

2.8.2 Interpreted vs Compiled Code

CL works with both *compiled* and *interpreted* code. When you type expressions at a Command Prompt, or use CL's `load` function to load `lisp` source files directly, you are introducing *interpreted* code into the system. This code does not undergo any optimization or translation into a lower-level machine representation — internally to CL it exists in a form very similar to its source code form. CL must do considerable work every time such code is evaluated, since it must be translated into lower-level machine instructions on each use.

When you create *compiled* code, on the other hand, for example by using CL's `compile-file` command in order to produce a `fasl` file, the code is optimized and translated into an efficient form. Loading the `fasl` file will replace any previous interpreted definitions with compiled definitions.

Interpreted code is the best to use for debugging a program, because the code has not undergone as many transformations, so the debugger can provide the programmer with more recognizable information.

If you are debugging with compiled code (e.g. after loading a `fasl` file), and you feel that not enough information is available from within the debugger, try redefining the function in interpreted form and running it again. To redefine a function in interpreted form you can, of course, simply `load` the `lisp` source file, rather than the compiled `fasl` file.

2.8.3 Use of (break) and C-c to interrupt

In order to cause a break in your program interactively, issue the `C-c` command. When running inside Emacs, you must use a "double" `C-c`, because the first `C-c` will be intercepted by Emacs. Depending on what your application is doing, it may take a few moments for it to respond to your break command. Once it responds, you will be given a normal debugger level, as described above.

[4] the *call stack* is the sequence of functions calls leading up to the error or break.

2.8.4 Profiling

CL has built-in functionality for monitoring its own performance. The most basic and commonly used component is the `time` Macro. You can "wrap" the `time` Macro around any expression. Functionally, `time` will not affect the behavior of the expression, but it will print out information about how long the expression took to be evaluated:

```
 CL-USER(2): (time (dotimes (n 10000) (* n 2)))
; cpu time (non-gc) 70 msec user, 0 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  70 msec user, 0 msec system
; real time  72 msec
```

It is often interesting to look at the difference in `time` between interpreted and compiled code. The example above is interpreted, since we typed it directly at the Command Line, so CL had no opportunity to optimize the `dotimes`. In compiled form, the above code consumes only one millisecond as opposed to 72 milliseconds.

In addition to the basic `time` Macro, Allegro CL provides a complete Profiler package, which allows you to monitor exactly what your program is doing, when it is doing it, how much time it takes, and how much memory it uses.

2.9 Developing Programs and Applications in CL

For developing a serious application in CL, you will want to have a convenient procedure in place for starting up your *development session*.

2.9.1 A Layered Approach

The development session consists of your favorite text editor with appropriate customizations, and a running CL process with any standard code loaded into it. While you can certainly develop serious applications on top of base CL, typically you will work in a *layered* fashion, on top of some standard tools or libraries, such as an embedded language, a webserver, etc.

Using one of the techniques outlined below, starting a customized development session with your standard desired tools should become convenient and transparent.

2.9.2 Compiling and Loading your Project

An individual project will generally consist of a *tree* of directories and files in your computer's filesystem. We will refer to this directory tree as the project's *codebase*.

Usually, it does not matter in what *order* the files in the codebase are compiled and loaded. Typical CL definitions, such as functions, classes, etc., can ordinarily be defined in any order. However, certain constructs, most notably Packages and Macros, *do* have order dependencies. In general, CL Packages and Macros must be defined before they can be used, or referenced.

To start a development session on your project, you will generally want CL to do the following:

1. Go through your project's Codebase, finding source files (in the proper order when necessary, as noted above)
2. For each file found, either load the corresponding binary file (if it is up-to-date), or compile the source file to create a new binary file and load it (if the source file has changed since the most recent compilation)

CL does not dictate one standard way of accomplishing this task, and in fact, several options are available. One option is to use a *defsystem* package, which will allow you to prepare a special file, similar to a "make" file, which lists out all your project's source files and their required load order. Allegro CL contains its own Defsystem package as an extension to CL, and the open-source MK:Defsystem is available through Sourceforge (`http://www.sourceforge.net`).

Using a Defsystem package requires you to maintain a catalog listing of all your project's source files. Especially in the early stages of a project, when you are adding, renaming, and deleting a lot of files and directories, maintaining this catalog listing by hand can become tiresome.

For this purpose, you may wish to use a lighter-weight utility for compiling and loading your files, at least in the early stages of development. For very small projects, you can use the Allegro CL function `excl:compile-file-if-needed`, an extension to CL, by writing a simple function which calls this function repeatedly to compile and load your project's files.

The Bootstrap package, available at `http://gdl.sourceforge.net`, provides the function `cl-lite`, which will traverse a codebase, compiling and loading files, and observing simple "ordering directive" files which can be placed throughout the codebase to enforce correct load ordering.

For a new project, the best option is probably to start with a simple but somewhat manual loading technique, then graduate into more automated techniques as they become necessary and as you become more familiar with the environment.

2.9.3 Creating an Application "Fasl" File

`Fasl` files in Allegro CL and most other CL systems have the convenient property that they can simply be concatenated, e.g. with the Unix/Linux `cat` command, to produce a *single* `fasl` file. Loading this single file will have the same effect as loading all the individual files which went into it, in the same order as they were concatenated.

In certain situations, this can be a convenient mechanism for loading and/or distributing your project. For example, you can prepare a concatenated `fasl` file, email it to a colleague, and the colleague needs only to load this file. Thus, there is no chance of confusion about how (e.g. in what order) to load individual files.

2.9.4 Creating an Image File

Loading Lisp source files or compiled binary files into memory at the start of each session can become slow and awkward for large programs.

An alternative approach is to build an *image file*. This entails starting a CL process, loading all application files into the running CL process, then creating a new "CL Image." In Allegro CL, this is done by invoking the function:

```
excl:dumplisp
```

The Image File is typically named with a `.dxl` extension. Once you have created an Image File, you can start a session simply by starting CL with that Image File, as follows:

```
lisp -I <image-file-name>.dxl
```

You must then compile/load any files that have changed since the Image File was created. Starting CL with the Image File has the same effect as starting a base CL and loading all your application files, but is simpler and much faster.

2.9.5 Building Runtime Images

CL Image files built with `excl:dumplisp` contain the entire CL environment, and may also contain information which is specific to a certain machine or certain site. Allegro CL also provides a mechanism for building *runtime images*, which contain only the information needed to run an end-user application. These Runtime Images can be packaged and distributed just as with any stand-alone software application.

Preparing runtime images for different platforms (e.g. Sun Solaris, Linux, Windows, etc.) is usually a simple matter of running a compilation and building a runtime image for the target platform — application source code typically remains identical among different platforms. See your platform-specific documentation for details about creating Runtime Images.

2.9.6 Using an Application Init File

In some cases, for a finished production application, you will want the CL process to execute certain commands upon startup (for example, loading some data from a database). However, you do not want to depend on the end-user having the appropriate `.clinit.cl` in his or her home directory. For such cases, you can specify a different location for the init file, which can be installed along with the application itself. One way to accomplish this is with the `-q` command-line argument to the `lisp` command. This argument will tell CL to look in the *current* directory, rather than in the default location, for the init file.

To use this technique, you would set up an "application home" directory, place the application's `.clinit.cl` file in this directory, and start the application with a production startup script. This script would first change into the "application home" directory, then start CL with your application's image file, or start a Runtime Image. Such a startup script might look like this:

```
cd $MYAPP_HOME
lisp -I image.dxl -q
```

In this example, both the application's init file and the application's image file would be housed in the directory named by the environment variable `$MYAPP_HOME`.

Chapter 3

The CL Language

If you are new to the CL language, we recommend that you supplement this chapter with other resources. See the Bibliography in Appendix B for some suggestions. The Bibliography also lists two interactive online CL tutorials from Tulane University and Texas A&M, which you may wish to visit.

In the meantime, this chapter will provide a condensed overview of the language. Please note, however, that this book is intended as a summary, and will not delve into some of the more subtle and powerful techniques possible with Common Lisp.

3.1 Overview of CL and its Syntax

The first thing you should observe about CL (and most languages in the Lisp family) is that it uses a generalized *prefix* notation.

One of the most frequent actions in a CL program, or at the toplevel *read-eval-print* loop, is to call a *function*. This is most often done by writing an *expression* which names the function, followed by its arguments. Here is an example:

```
(+ 2 2)
```

This expression consists of the function named by the symbol "+," followed by the arguments `2` and another `2`. As you may have guessed, when this expression is evaluated it will return the value `4`.

Try it: Try typing this expression at your command prompt, and see the return-value being printed on the console.

What is it that is actually happening here? When CL is asked to *evaluate* an *expression* (as in the toplevel *read-eval-print* loop), it evaluates the expression according to the following rules:

1. If the expression is a *number* (i.e. looks like a number), it simply evaluates to itself (a number):

   ```
   CL-USER(8): 99
   99
   ```

2. If the expression looks like a *string* (i.e. is surrounded by double-quotes), it also simply evaluates to itself:

```
CL-USER(9): "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. If the expression looks like a literal *symbol*, it will simply evaluate to that *symbol* (more on this in Section 3.1.4):

```
CL-USER(12): 'my-symbol
MY-SYMBOL
```

4. If the expression looks like a *list* (i.e. is surrounded by parentheses), CL assumes that the *first* element in this list is a *symbol* which names a *function* or a *macro*, and the *rest* of the elements in the list represent the *arguments* to the function or macro. (We will discuss *functions* first, *macros* later). A *function* can take zero or more arguments, and can *return* zero or more *return-values*. Often a function only returns one return-value:

```
CL-USER(14): (expt 2 5)
32
```

Try it: Try typing the following functional expressions at your command prompt, and convince yourself that the printed return-values make sense:

```
(+ 2 2)

(+ 2)

2

(+)

(+ (+ 2 2) (+ 3 3))

(+ (+ 2 2))

(sys:user-name)

(user-homedir-pathname)

(get-universal-time) ;; returns number of seconds since January 1, 1900
```

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

"I am Dr. Strangelove"

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

3.1.1 Evaluation of Arguments to a Function

Note that the arguments to a function can *themselves* be any kind of the above expressions. They are evaluated in order, from left to right, and finally they are passed to the function for it to evaluate. This kind of nesting can be arbitrarily deep. Do not be concerned about getting lost in an ocean of parentheses — most serious text editors can handle deeply nested parentheses with ease, and moreover will automatically indent your expressions so that you, as the programmer, never have to concern yourself with matching parentheses.

3.1.2 Lisp Syntax Simplicity

One of the nicest things about Lisp is the simplicity and consistency of its syntax. What we have covered so far pertaining to the evaluation of arguments to functions is *just about everything you need to know* about the syntax. The rules for macros are very similar, with the primary difference being that all the arguments of a macro are not necessarily evaluated at the time the macro is called — they can be transformed into something else first.

This simple consistent syntax is a welcome relief from other widely used languages such as C, C++, Java, Perl, and Python. These languages claim to use a "more natural" "infix" syntax, but in reality their syntax is a confused mixture of prefix, infix, and postfix. They use infix only for the simplest arithmetic operations such as

```
2 + 2
```

and

```
3 * 13
```

and use postfix in certain cases such as

```
i++
```

and

```
char*
```

But mostly they actually use a prefix syntax much the same as Lisp, as in:

```
split(@array, ":");
```

So, if you have been programming in these other languages, you have been using prefix notation all along, but may not have noticed it. If you look at it this way, the prefix notation of Lisp will seem much less alien to you.

3.1.3 Turning Off Evaluation

Sometimes you want to specify an expression at the toplevel *read-eval-print* loop, or inside a program, but you do not want CL to evaluate this expression. You want just the literal expression. CL provides the special operator `quote` for this purpose. For example,

```
(quote a)
```

will return the literal symbol `A`. Note that, by default, the CL reader will convert all symbol names to uppercase at the time the symbol is read into the system. Note also that symbols, if evaluated "as is," (i.e. without the `quote`), will by default behave as *variables*, and CL will attempt to retrieve an associated *value*. More on this later.

Here is another example of using `quote` to return a literal expression:

```
(quote (+ 1 2))
```

Unlike evaluating a list expression "as is," this will return the literal list `(+ 1 2)`. This list may now be accessed as a normal list data structure[1].

Common Lisp defines the abbreviation `'` as a shorthand way to "wrap" an expression in a call to `quote`. So the previous examples could equivalently be written as:

```
'a

'(+ 1 2)
```

3.1.4 Fundamental CL Data Types

Common Lisp natively supports many data types common to other languages, such as numbers, strings, and arrays. Native to CL is also a set of types which you may not have come across in other languages, such as lists, symbols, and hash tables. In this overview we will touch on numbers, strings, symbols, and lists. Later in the book we will provide more detail on some CL-specific data types.

Regarding data types, CL follows a paradigm called *dynamic* typing. Basically this means that *values* have type, but *variables* do not necessarily have type, and typically variables are not "pre-declared" to be of a particular type.

Numbers

Numbers in CL form a hierarchy of types, which includes *Integers*, *Ratios*, *Floating Point*, and *Complex* numbers. For many purposes you only need to think of a value as a "number," without getting any more specific than that. Most arithmetic operations, such as `+`, `-`, `*`,

[1] It should be noted, however, that when a quoted literal list is created like this, it is now technically a constant — you should not modify it with destructive operators.

`/`, etc, will automaticaly do any necessary type coercion on their arguments and will return a number of the appropriate type.

CL supports a full range of floating-point decimal numbers, as well as true *Ratios*, which means that `1/3` is a true one-third, not `0.333333333` rounded off at some arbitrary precision.

As we have seen, numbers in CL are a native data type which simply evaluate to themselves when entered at the toplevel or included in an expression.

Strings

Strings are actually a specialized kind of *Array*, namely a one-dimensional array (vector) made up of characters. These characters can be letters, numbers, or punctuation, and in some cases can include characters from international character sets (e.g. Unicode) such as Chinese Hanzi or Japanese Kanji. The string delimiter in CL is the double-quote character (`"`).

As we have seen, strings in CL are a native data type which simply evaluate to themselves when included in an expression.

Symbols

Symbols are such an important data structure in CL that people sometimes refer to CL as a "Symbolic Computing Language." Symbols are a type of CL object which provides your program with a built-in mechanism to store and retrieve values and functions, as well as being useful in their own right. A symbol is most often known by its name (actually a string), but in fact there is much more to a symbol than its name. In addition to the name, symbols also contain a *function* slot, a *value* slot, and an open-ended *Property-list* slot in which you can store an arbitrary number of named properties.

For a named function such as `+`, the function-slot of the symbol `+` contains the actual function itself. The value-slot of a symbol can contain any value, allowing the symbol to act as a global variable, or *Parameter*. And the Property-list, or *plist* slot, can contain an arbitrary amount of information.

This separation of the symbol data structure into function, value, and plist slots is one obvious distinction between Common Lisp and most other Lisp dialects. Most other dialects allow only one (1) "thing" to be stored in the symbol data structure, other than its name (e.g. either a function or a value, but not both at the same time). Because Common Lisp does not impose this restriction, it is not necessary to contrive names, for example for your variables, to avoid conflicting with existing "reserved words" in the system. For example, "`list`" is the name of a built-in function in CL. But you may freely use "`list`" as a variable as well. There is no need to contrive arbitrary abbreviations such as "`lst`."

How symbols are evaluated depends on where they occur in an expression. As we have seen, if a symbol appears *first* in a list expression, as with the `+` in `(+ 2 2)`, the symbol is evaluated for its function slot. If the first element of an expression is a symbol which indeed does contain a function in its function slot, then any symbol which appears *anywhere else* (i.e. in the *rest*) of the expression is taken as a variable, and it is evaluated for its global or local value, depending on its *scope*. More on variables and scope later.

As noted in Section 3.1.3, if you want a literal symbol itself, one way to achieve this is to "quote" the symbol name:

```
'a
```

Another way is for the symbol to appear within a quoted list expression:

```
'(a b c)

'(a (b c) d)
```

Note that the quote (`'`) applies across everything in the list expression, including any sub-expressions.

Lists

Lisp takes its name from its strong support for the list data structure. The list concept is important to CL for more than this reason alone — lists are important because *all CL programs themselves are lists!* Having the list as a native data structure, as well as the form of all programs, means that it is especially straightforward for CL programs to compute and generate other CL programs. Likewise, CL programs can read and manipulate other CL programs in a natural manner. *This cannot be said of most other languages, and is one of the primary distinguishing characteristics of CL.*

Textually, a list is defined as zero or more elements surrounded by parentheses. The elements can be objects of any valid CL data types, such as numbers, strings, symbols, lists, or other kinds of objects. As we have seen, you must quote a literal list to evaluate it or CL will assume you are calling a function.

Now look at the following list:

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

This list also happens to be a valid CL program (function definition, in this case). Don't worry about analyzing the function right now, but do take a few moments to convince yourself that it meets the requirements for a list. What are the types of the elements in this list?

In addition to using the quote (`'`) to produce a literal list, another way to produce a list is to call the function `list`. The function `list` takes any number of arguments, and returns a list made up from the result of evaluating each argument (as with all functions, the arguments to the `list` function get evaluated, from left to right, before being passed into the function). For example,

```
(list 'a 'b (+ 2 2))
```

will return the list `(A B 4)`. The two quoted symbols evaluate to symbols, and the function call `(+ 2 2)` evaluates to the number `4`.

3.1.5 Functions

Functions form the basic building block of CL. Here we will give a brief overview of how to define a function; later we will go into more detail on what a function actually is.

A common way to define named functions in CL is with the macro `defun`, which stands for DEFinition of a FUNction. `Defun` takes as its arguments a symbol, an *argument list*, and a *body*:

```
(defun my-first-lisp-function ()
  (list 'hello 'world))
```

Because `defun` is a *macro*, rather than a function, it does not follow the hard-and-fast rule that all its arguments are evaluated as expressions — specifically, the symbol which names the function does not have to be quoted, nor does the argument list. These are taken as a literal symbol and a literal list, respectively.

Once the function has been defined with `defun`, you can call it just as you would call any other function, by wrapping it in parentheses together with its arguments:

```
CL-USER(56): (my-first-lisp-function)
(HELLO WORLD)

CL-USER(57): (defun square(x)
               (* x x))
SQUARE

CL-USER(58): (square 4)
16
```

Declaring the types of the arguments to a function is not required.

3.1.6 Global and Local Variables

Global Variables

Global variables in CL are usually also known as *special* variables. They can be established by calling `defparameter` or `defvar` from the *read-eval-print* loop, or from within a file that you compile and load into CL:

```
(defparameter *todays-temp* 90)
(defvar *humidity* 70)
```

The surrounding asterisks on these symbol names are part of the symbol names themselves, and have no significance to CL. This is simply a long-standing naming convention for global variables (i.e. parameters), to make them easy to pick out with the human eye.

`Defvar` and `defparameter` differ in one important way: if `defvar` is evaluated with a symbol which already has a global value, it will *not* overwrite it with a new value. `Defparameter`, on the other hand, *will* overwrite it with a new value:

```
CL-USER(4): *todays-temp*
90

CL-USER(5): *todays-humidity*
70

CL-USER(6): (defparameter *todays-temp* 100)
*TODAYS-TEMP*

CL-USER(7): (defvar *todays-humidity* 50)
*TODAYS-HUMIDITY*

CL-USER(8): *todays-temp*
100

CL-USER(9): *todays-humidity*
70
```

`*todays-humidity*` did not change because we used `defvar`, and it already had a previous value.

The value for any Parameter can always be changed from the toplevel with setq:

```
CL-USER(11): (setq *todays-humidity* 30)
30

CL-USER(12): *todays-humidity*
30
```

Although `setq` will work with new variables, stylistically it should only be used with already-established variables.

During application development, it often makes sense to use `defvar` rather than `defparameter` for variables whose values might change during the running and testing of the program. This way, you will not unintentionally reset the value of a parameter if you compile and load the source file where it is defined.

Local Variables

New local variables can be introduced with the macro `let`:

```
CL-USER(13): (let ((a 20)
                   (b 30))
               (+ a b))
50
```

As seen in the above example, `let` takes an *assignment section* and a *body*. `Let` is a macro, rather than a function[2], so it does not follow the hard-and-fast rule that all its arguments are evaluated. Specifically, the assignment section

[2]See Section 3.8.5 for a discussion of how to recognize macros and functions

```
((a 20)
 (b 30))
```

is not evaluated (actually, the macro has the effect of transforming this into something else before the CL evaluator even sees it). Because this assignment section is not evaluated by the `let` macro, it does not have to be quoted, like a list which is an argument to a function would have to be. As you can see, the assignment section is a list of lists, where each internal list is a pair whose `first` is a symbol, and whose `second` is a value (which *will* be evaluated). These symbols (`a` and `b`) are the local variables, and they are assigned to the respective values.

The *body* consists of any number of expressions which come after the assignment section and before the closing parenthesis of the `let` statement. The expressions in this body are evaluated normally, and of course any expression can refer to the value of any of the local variables simply by referring directly to its symbol. If the body consists of more than one expression, the final return-value of the body is the return-value of its last expression.

New Local variables in CL are said to have *lexical* scope, which means that they are only accessible in the code which is textually contained within the body of the `let`. The term *lexical* is derived from the fact that the behavior of these variables can be determined simply by *reading* the text of the source code, and is not affected by what happens during the program's execution.

Dynamic scope happens when you basically mix the concept of global parameters and local `let` variables. That is, if you use the name of a previously established parameter inside the assignment section of a `let`, like this:

```
(let ((*todays-humidity* 50))
  (do-something))
```

the parameter will have the specified value not only textually within the the body of the `let`, *but also* in any functions which may get called from within the body of the let. The global value of the parameter in any other contexts will not be affected. Because this scoping behavior is determined by the runtime "dynamic" execution of the program, we refer to it as dynamic scope.

Dynamic scope is often used for changing the value of a particular global parameter only within a particular tree of function calls. Using Dynamic scope, you can accomplish this without affecting other places in the program which may be referring to the parameter. Also, you do not have to remember to have your code "reset" the parameter back to a default global value, since it will automatically "bounce" back to its normal global value.

Dynamic scoping capability is especially useful in a *multithreaded* CL, i.e., a CL process which can have many (virtually) simultaneous threads of execution. A parameter can take on a dynamically scoped value in one thread, without affecting the value of the parameter in any of the other concurrently running threads.

3.2 The List as a Data Structure

In this section we will present some of the fundamental native CL operators for manipulating lists as data structures. These include operators for doing things such as:

1. finding the length of a list
2. accessing particular members of a list
3. appending multiple lists together to make a new list
4. extracting elements from a list to make a new list

3.2.1 Accessing the Elements of a List

Common Lisp defines the accessor functions `first` through `tenth` as a means of accessing the first ten elements in a list:

```
CL-USER(5): (first '(a b c))
A

CL-USER(6): (second '(a b c))
B

CL-USER(7): (third '(a b c))
C
```

For accessing elements in an arbitrary position in the list, you can use the function `nth`, which takes an integer and a list as its two arguments:

```
CL-USER(8): (nth 0 '(a b c))
A

CL-USER(9): (nth 1 '(a b c))
B

CL-USER(10): (nth 2 '(a b c))
C

CL-USER(11): (nth 12 '(a b c d e f g h i j k l m n o p))
M
```

Note that `nth` starts its indexing at *zero* (0), so `(nth 0 ...)` is equivalent to `(first ...)`, and `(nth 1 ...)` is equivalent to `(second ...)`, etc.

3.2.2 The "Rest" of the Story

A very common operation in CL is to perform some operation on the `first` of a list, then perform the same operation on each `first` of the `rest` of the list, repeating the procedure until the end of the list, i.e. the Empty list, is reached. The function `rest` is very helpful in such cases:

```
CL-USER(59): (rest '(a b c))
(B C)
```

As you can see from this example, `rest` returns a list consisting of all but the `first` of its argument.

3.2.3 The Empty List

The symbol `NIL` is defined by Common Lisp to be equivalent to the Empty List, `()`. `NIL` also has the interesting property that its value is *itself*, which means that it will always evaluate to itself, whether or not it is quoted. So `NIL`, `'NIL`, and `()` all evaluate to the Empty List, whose default printed representation is `NIL`:

```
CL-USER(14): nil
NIL

CL-USER(15): 'nil
NIL

CL-USER(16): ()
NIL
```

The function `null` can be used to test whether or not something is the Empty List:

```
CL-USER(17): (null '(a b c))
NIL

CL-USER(18): (null nil)
T

CL-USER(19): (null ())
T
```

As you may have deduced from the above examples, the symbol `T` is CL's default representation for "true." As with `NIL`, `T` evaluates to itself.

3.2.4 Are You a List?

To test whether or not a particular object is a list, CL provides the function `listp`. Like many other functions which end with the letter "`p`," this function takes a single argument and checks whether it meets certain criteria (in this case, whether it qualifies as a list). These *predicate* functions ending in the letter "`p`" will always return either `T` or `NIL`:

```
CL-USER(20): (listp '(pontiac cadillac chevrolet))
T

CL-USER(21): (listp 99)
NIL

CL-USER(22): (listp nil)
T
```

Note that `(listp nil)` returns `T`, since `NIL` is indeed a list (albeit the empty one).

3.2.5 The conditional `If`

Before continuing with a number of other basic list functions, we will cover the macro `if`, which allows simple conditionals. `If` takes three arguments, a *test-form*, a *then-form*, and an *else-form*. When an `if` form is evaluated, it first evaluates its test-form. If the form returns non-`NIL`, it will evaluate the then-form, else it will evaluate the else-form. The nesting of multiple `if` expressions is possible, but not advised; later we will cover other constructs which are more appropriate for such cases.

Here are some simple examples using the `if` operator:

```
CL-USER(2): (if (> 3 4)
                "no"
              "yes")
"yes"


CL-USER(3): (if (listp '("Chicago" "Detroit" "Toronto"))
                "it is a list"
              "it ain't a list")
"it is a list"

CL-USER(4): (if (listp 99)
                "it is a list"
              "it ain't a list")
"it ain't a list"
```

3.2.6 Length of a List

Normally you can use the function `length` to get the number of elements in a list as an integer:

```
CL-USER(5): (length '(gm ford chrysler volkswagen))
4
```

```
CL-USER(6): (length nil)
0
```

The `length` function, as with most of Common Lisp, can itself be implemented in CL. Here is a simplified version of an `our-length` function which illustrates how `length` might be implemented:

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

Note that the function uses the symbol `list` to name its argument, which is perfectly valid as we discussed in section 3.1.4.

In English, this function says basically: "If the list is empty, its length is zero. Otherwise its length is one greater than the length of its `rest`." As with many functions which operate on lists, this *recursive* definition is a natural way to express the `length` operation.

3.2.7 Member of a List

The `member` function will help you to determine whether a particular item is an element of a particular list. Like many similar functions, `member` uses `eql` to test for equality, which is one of the most basic equality functions in CL (see Section 3.8.4 for a discussion of equality in CL). `Eql` basically means the two objects must be the same symbol, integer, or actual object (i.e. the same address in memory).

Member takes two arguments: an item and a list. If the item is not in the list, it returns `NIL`. Otherwise, it returns the `rest` of the list, starting from the found element:

```
CL-USER(7): (member 'dallas '(boston san-francisco portland))
NIL

CL-USER(8): (member 'san-francisco '(boston san-francisco portland))
(SAN-FRANCISCO PORTLAND)
```

As with `length`, we could define `member` using a function definition which is *very* close to the English description of what the function is supposed to do[3]:

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

[3]The use of the "nested" `if` statement in this example, while functionally correct, is a violation of good CL style. Later we will learn how to avoid nested `if` statements.

In English, you might read this function to say "If the list is empty, return `NIL`. Otherwise, if the desired item is the same as the `first` of the list, return the entire list. Otherwise, do the same thing on the `rest` of the list."

Note that, for the purposes of any kind of logical operators, returning *any* non-`NIL` value is "as good as" returning `T`. So the possible return-values of the `member` function are as good as returning `T` or `NIL` as far as any logical operators are concerned.

3.2.8 Getting Part of a List

`Subseq` is a common function to use for returning a portion of a list (or actually any type of *Sequence*). Subseq takes at least two arguments, a list and an integer indicating the position to *start* from. It also takes an optional third argument, an integer indicating the position to *stop*. Note that the position indicated by this third argument is *not* included in the returned sub-list:

```
CL-USER(9): (subseq '(a b c d) 1 3)
(B C)

CL-USER(10): (subseq '(a b c d) 1 2)
(B)

CL-USER(11): (subseq '(a b c d) 1)
(B C D)
```

Note also that the optional third argument in effect defaults to the `length` of the list.

3.2.9 Appending Lists

The function `append` takes any number of lists, and returns a new list which results from appending them together. Like many CL functions, `append` does not *side-effect*, that is, it simply returns a new list as a return-value, but does not modify its arguments in any way:

```
CL-USER(6): (setq my-slides '(introduction welcome lists functions))
(INTRODUCTION WELCOME LISTS FUNCTIONS)

CL-USER(7): (append my-slides '(numbers))
(INTRODUCTION WELCOME LISTS FUNCTIONS NUMBERS)

CL-USER(8): my-slides
(INTRODUCTION WELCOME LISTS FUNCTIONS)

CL-USER(9): (setq my-slides (append my-slides '(numbers)))
(INTRODUCTION WELCOME LISTS FUNCTIONS NUMBERS)

CL-USER(10): my-slides
(INTRODUCTION WELCOME LISTS FUNCTIONS NUMBERS)
```

Note that the simple call to `append` does not affect the variable `my-slides`. If we wish to modify the value of this variable, however, one way to do this is by using `setq` in combination with the call to `append`. Note also the use of `setq` directly at the toplevel with a new variable name. For testing and "hacking" at the command prompt, this is acceptable, but in a finished program all such global variables should really be formally declared with `defparameter` or `defvar`.

3.2.10 Adding Elements to a List

To add a single element to the front of a list, you can use the function `cons`:

```
CL-USER(13): (cons 'a '(b c d))
(A B C D)
```

`Cons` is actually the low-level primitive function upon which many of the other list-constructing functions are built. As you may have guessed, "`cons`" stands for "CONStruct," as in "construct a list." When you read or hear people talking about a CL program doing a lot of "consing," they are referring to the program's behavior of building a lot of list structures, many of which are transitory and will need to be "freed up" by the automatic memory management subsystem, or "garbage collector."

As with `append`, `cons` is *non-destructive*, meaning it does no side-effecting (modification) to its arguments.

3.2.11 Removing Elements from a List

The function `remove` takes two arguments, any item and a list, and returns a new list with all occurences of the item taken out of it:

```
CL-USER(15): (setq data '(1 2 3 1000 4))
(1 2 3 1000 4)

CL-USER(16): (remove 1000 data)
(1 2 3 4)

CL-USER(17): data
(1 2 3 1000 4)

CL-USER(18): (setq data (remove 1000 data))
(1 2 3 4)

CL-USER(19): data
(1 2 3 4)
```

Like `append` and `cons`, `remove` is non-destructive and so does not modify its arguments. As before, one way to achieve modification with variables is to use `setq`.

3.2.12 Sorting Lists

The function `sort` will sort any sequence (including, of course, a list), based on comparing the elements of the sequence using any applicable comparison function, or *predicate* function. Because of efficiency reasons, the designers of CL made the decision to allow `sort` to modify ("recycle" the memory in) its sequence argument — so you must always "catch" the result of the sorting by doing something explicitly with the return-value:

```
CL-USER(20): (setq data '(1 3 5 7 2 4 6))
(1 3 5 7 2 4 6)

CL-USER(21): (setq data (sort data #'<))
(1 2 3 4 5 6 7)

CL-USER(22): (setq data (sort data #'>))
(7 6 5 4 3 2 1)
```

Notice that the comparison function must be a function which can take two arguments — any representative two elements in the given sequence — and compare them to give a `T` or `NIL` result. The "`#'`" notation is a shorthand way of retrieving the actual function object associated with a symbol or expression (in this case, the `<` or `>` symbol). We will cover more on this in Section 3.4.

In the above examples, we simply reset the value of the variable `data` to the result of the sort, which is a common thing to do. If one wanted to retain the original pre-sorted sequence, a "safe" version of `sort` could be defined as follows:

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

3.2.13 Treating a List as a Set

The functions `union`, `intersection`, and `set-difference` take two lists and compute the corresponding set operation. Because mathematical sets have no notion of ordering, the order of the results returned by these functions is purely arbitrary, so you should never depend on the results of these set operations being in any particular order:

```
CL-USER(23): (union '(1 2 3) '(2 3 4))
(1 2 3 4)

CL-USER(25): (intersection '(1 2 3) '(2 3 4))
(3 2)

CL-USER(26): (set-difference '(1 2 3 4) '(2 3))
(4 1)
```

3.2.14 Mapping a Function to a List

If you have one list, and desire another list of the same length, there is a good chance that you can use one of the mapping functions. `Mapcar` is the most common of such functions. `Mapcar` takes a function and one or more lists, and *maps* the function across each element of the list, producing a new resulting list. The term *car* comes from the original way in Lisp of referring to the `first` of a list ("contents of address register"). Therefore `mapcar` takes its function and applies it to the `first` of each successive `rest` of the list:

```
CL-USER(29): (defun twice (num)
               (* num 2))
TWICE

CL-USER(30): (mapcar #'twice '(1 2 3 4))
(2 4 6 8)
```

"Lambda" (unnamed) functions are used *very* frequently with `mapcar` and similar mapping functions. More on this in Section 3.4.3.

3.2.15 Property Lists

Property lists ("*plists*") provide a simple yet powerful way to handle *keyword-value pairs*. A plist is simply a list, with an even number of elements, where each pair of elements represents a named value. Here is an example of a plist:

```
(:michigan "Lansing" :illinois "Springfield"
 :pennsylvania "Harrisburg")
```

In this plist, the *keys* are keyword symbols, and the *values* are strings. The keys in a plist are very often keyword symbols. *keyword symbols* are symbols whose names are preceded by a colon (`:`), and which are generally used just for matching the symbol itself (i.e. typically they are not used for their symbol-value or symbol-function).

For accessing members of a plist, CL provides the function `getf`, which takes a plist and a Key:

```
CL-USER(34): (getf '(:michigan "Lansing"
                     :illinois "Springfield"
                     :pennsylvania "Harrisburg")
                   :illinois)
"Springfield"
```

3.3 Control of Execution

"Control of execution" in CL boils down to "control of evaluation." Using some standard and common macros, you control which forms get evaluated, among certain choices.

```
(if (eql status :all-systems-go)
    (progn (broadcast-countdown) (flash-colored-lights)
           (play-eerie-music)(launch-rocket))
  (progn (broadcast-apology) (shut-down-rocket)))
```

Figure 3.1: Progn used with If

3.3.1 If

Perhaps the most basic operator for imposing control is the `if` operator, which we have already introduced briefly in Section 3.2.5. The `If` operator takes exactly three arguments, each of which is an expression which may be evaluated. The first is a *test-form*, the second is a *then-form*, and the third is an *else-form* (which may be left out for a default of `NIL`). When CL evaluates the `if` form, it will first evaluate the `test-form`. Depending on whether the return-value of the test-form is non-`NIL` or `NIL`, either the then-form or the else-form will be evaluated.

If you want to group multiple expressions together to occur as part of the then-form or the else-form, you must wrap them in an enclosing block. `Progn` is commonly used for this purpose. `Progn` will accept any number of forms as arguments, and will evaluate each of them in turn, finally returning the return-value of the last of them (see Figure 3.1).

3.3.2 When

In some ways, `when` is similar to `if`, but you should use `when` in cases where you know that the else-clause is simply `NIL`. This turns out to be a common situation. Another advantage of using `when` in these cases is that there is no need for `progn` to group multiple forms — `when` simply takes a Test-form then any number of "Then-forms:"

```
CL-USER(36): (when (eql database-connection :active)
               (read-record-from-database)
               (chew-on-data-from-database)
               (calculate-final-result))
999

CL-USER(37): (when (> 3 4)
               (princ "I don't think so..."))
NIL
```

(Note that the database functions above are examples only and are not pre-defined in CL).

3.3.3 Logical Operators

The Logical Operators `and` and `or` evaluate one or more expressions given as arguments, depending on the return-values of the expressions. As we covered in Section 3.2.3, `T` is

CL's default representation for "True," NIL (equivalent to the Empty List) is CL's default representation for "False," and any non-NIL value is "as good as" T as far as "Truth" is concerned:

```
CL-USER(38): (and (listp '(chicago detroit boston new-york))
                  (listp 3.1415))
NIL

CL-USER(39): (or (listp '(chicago detroit boston new-york))
                 (listp 3.1415))
T
```

In fact what is happening here is that the operator **and** evaluates its arguments (expressions) one at a time, from left to right, returning NIL as soon as it finds one which returns NIL — otherwise it will return the value of its last expression. Or evaluates its arguments until it finds one which returns *non*-NIL, and returning that *non*-NIL return-value if it finds one. Otherwise it will end up returning NIL.

Not takes a single expression as an argument and negates it — that is, if the expression returns NIL, not will return T, and if the argument returns non-NIL, not will return NIL. Logically, not behaves identically with the function null — but semantically, null carries the meaning of "testing for an Empty List," while not carries the meaning of logical negation:

```
CL-USER(45): (not NIL)
T

CL-USER(46): (not (listp 3.1415))
T
```

3.3.4 Cond

Because the use of "nested" if statements is not appropriate, CL provides the macro cond to accomodate this. Use cond in situations when in other languages you might use a repeated "If...then...else if...else if...else if..." Cond takes as arguments any number of *test-expression* forms. Each of these forms consists of a list whose first is an expression to be evaluated, and whose rest is any number of expressions which will be evaluated if the first form evaluates to non-NIL. As soon as a non-NIL form is found, its corresponding expressions are evaluated and the entire cond is finished (i.e. there is no need for an explicit "break" statement or anything of that nature):

```
CL-USER(49): (let ((n 4))
               (cond ((> n 10) "It's a lot")
                     ((= n 10) "It's kind of a lot")
                     ((< n 10) "It's not a lot")))
"It's not a lot"
```

Because T as an expression simply evaluates to itself, a convenient way to specify a "default" or "catch-all" clause in a cond is to use T as the final conditional expression:

```
CL-USER(50): (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"
```

3.3.5 Case

If you want to make a decision based on comparing a variable against a known set of constant values, case is often more concise than cond:

```
CL-USER(51): (let ((color :red))
               (case color
                 (:blue "Blue is okay")
                 (:red  "Red is actually her favorite color")
                 (:green "Are you nuts?")))
"Red is actually her favorite color"
```

Note that case uses eql to do its matching, so it will only work with constants which are symbols (including keyword symbols), integers, etc. It will not work with strings.

CL also provides the macros ecase and ccase, which work just like case but provide some automatic error-handling for you in cases when the given value does not match any of the keys.

The symbol otherwise has special meaning to the case macro — it indicates a "default," or "catch-all" case:

```
CL-USER(52): (let ((color :orange))
               (case color
                 (:blue "Blue is okay")
                 (:red  "Red is actually her favorite color")
                 (:green "Are you nuts?")
                 (otherwise "We have never heard of that!")))
"We have never heard of that!"
```

3.3.6 Iteration

In addition to its innate abilities to do recursion and mapping, CL provides several operators for doing traditional iteration ("looping"). These include macros such as do, dolist, dotimes, and the "everything-including-the-kitchen-sink" iterating macro, loop. Dolist and dotimes are two of the most commonly used ones, so we will cover them here:

```
CL-USER(53): (let ((result 0))
               (dolist (elem '(4 5 6 7) result)
```

```
                (setq result (+ result elem))))
22
```

As seen above, `dolist` takes an *initialization-list* and a *body*. The Initialization-list consists of an *iterator-variable*, a *list-form*, and an optional *return-value-form* (which defaults to `NIL`). The body will be evaluated once with the iterator-variable set to each element in the list which is returned by the list-form. When the list has been exhausted, the return-value-form will be evaluated and this value will be returned from the entire `dolist`. In other words, the number of iterations of a `dolist` is driven by the length of the list returned by its list-form.

`Dotimes` is similar in many ways to `dolist`, but instead of a list-form, it takes an expression which should evaluate to an integer. The body is evaluated once with the iterator-variable set to each integer starting from zero (0) up to *one less than* the value of the Integer-form:

```
CL-USER(54): (let ((result 1))
               (dotimes (n 10 result)
                 (setq result (+ result n))))
46
```

In situations when you want the program to exit early from an iteration, you can usually accomplish this by calling `return` explicitly.

3.4 Functions as Objects

3.4.1 Named Functions

When working with CL, it is important to understand that a function is actually a kind of object, separate from any symbol or symbol-name with which it might be associated. You can access the actual function-object in a symbol's function-slot by calling the function `symbol-function` on a symbol, like this:

```
(symbol-function '+)
#<Function +>
```

or this:

```
(symbol-function 'list)
#<Function LIST>
```

As you can see, the *printed representation* of a named function-object contains the function's symbol-name enclosed in pointy brackets, preceded by a pound sign.

You can explicitly call a function-object with the function *funcall*:

```
CL-USER(10): (funcall (symbol-function '+) 1 2)
3
```

```
CL-USER(11): (setq my-function (symbol-function '+))
#<Function +>

CL-USER(12): (funcall my-function 1 2)
3
```

A shorthand way of writing `symbol-function` is to write `#'` before the name of a function:

```
CL-USER(10): (funcall #'+ 1 2)
3

CL-USER(11): (setq my-function #'+)
#<Function +>

CL-USER(12): (funcall my-function 1 2)
3
```

3.4.2 Functional Arguments

Functional arguments are arguments to functions which *themselves* are function-objects. We have seen this already with `mapcar`, `sort`, and `funcall`:

```
CL-USER(13): (defun add-3 (num)
                (+ num 3))
ADD-3

CL-USER(14): (symbol-function 'add-3)
#<Interpreted Function ADD-3>

CL-USER(15): (mapcar #'add-3 '(2 3 4))
(5 6 7)

CL-USER(17): (funcall #'add-3 2)
5
```

The idea of passing functions around to other functions is sometimes referred to as `higher-order functions`. CL provides natural support for this concept.

3.4.3 Anonymous Functions

Sometimes a function will be used so seldom that it is hardly worth going to the extent of creating Named function with `defun`. For these cases, CL provides the concept of the *lambda*, or "anonymous" (unnamed) function. To use a lambda function, you place the entire function definition in the position where you normally would put just the symbol which names the function, identified with the special symbol `lambda`:

```
CL-USER(19): (mapcar #'(lambda(num) (+ num 3))
                     '(2 3 4))
(5 6 7)

CL-USER(20): (sort '((4 "Buffy") (2 "Keiko") (1 "Judy") (3 "Aruna"))
                   #'(lambda(x y)
                   (< (first x) (first y))))
((1 "Judy") (2 "Keiko") (3 "Aruna") (4 "Buffy"))
```

3.4.4 Optional Arguments

Optional arguments to a function must be specified after any required arguments, and are identified with the symbol `&optional` in the argument list. Each optional argument can be either a symbol or a list containing a symbol and an expression returning a default value. In the case of a symbol, the default value is taken to be `NIL`:

```
CL-USER(23): (defun greeting-list (&optional (username "Jake"))
                (list 'hello 'there username))
GREETING-LIST

CL-USER(24): (greeting-list)
(HELLO THERE "Jake")

CL-USER(25): (greeting-list "Joe")
(HELLO THERE "Joe")
```

3.4.5 Keyword Arguments

Keyword arguments to a function are basically *named* optional arguments. They are defined much the same as optional arguments:

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

But when you call a function with keyword arguments, you "splice in" what amounts to a plist containing keyword symbols for the names of the arguments along with their values:

```
CL-USER(27): (greeting-list :greeting "how have you been?")
(HELLO THERE "Jake" "how have you been?")

CL-USER(28): (greeting-list :greeting "how have you been?" :username "Joe")
(HELLO THERE "Joe" "how have you been?")
```

Note that with keyword arguments, you use a normal symbol (i.e. *without* the preceding colon) in the *definition* of the function, but a keyword symbol as the "tag" when *calling* the function. There is a logical reason behind this, which you will understand soon.

3.5 Input, Output, Streams, and Strings

Input and Output in CL is done through objects called *streams*. A stream is a source or destination for pieces of data which generally is connected to some physical source or destination, such as a terminal console window, a file on the computer's hard disk, or a web browser. Each thread of execution in CL defines global parameters which represent some standard Streams:

- `*standard-input*` is the default data source.
- `*standard-output*` is the default data destination.
- `*terminal-io*` is bound to the console (command prompt) window, and by default this is identical to both `*standard-input*` and `*standard-output*`.

3.5.1 Read

`Read` is the standard mechanism for reading data items into CL. `Read` takes a single optional argument for its stream, and reads a single data item from that stream. The default for the stream is `*standard-input*`. If you simply type `(read)` at the command prompt, CL will wait for you to enter some valid Lisp item, and will return this item. `Read` simply reads, it does *not* evaluate, so there is no need to quote the data being read. You can set a variable to the result of a call to read:

```
(setq myvar (read))
```

The function `read-line` is similar to `read`, but will read characters until it encounters a new line or end-of-file, and will return these characters in the form of a string. `Read-line` is appropriate for reading data from files which are in a "line-based" format rather than formatted as Lisp expressions.

3.5.2 Print and Prin1

`Print` and `prin1` each take exactly one CL object and output its printed representation to a stream, which defaults to `*standard-output*`. `Print` puts a space and a newline after the item (effectively separating individual items with whitespace), and `Prin1` outputs the item with no extra whitespace:

```
CL-USER(29): (print 'hello)

HELLO
HELLO
```

In the above example, you see the symbol `HELLO` appearing twice: the first time is the output actually being printed on the console, and the second is the normal return-value of the call to `print` being printed on the console by the *read-eval-print* loop. `Print` and `Prin1` both print their output *readably*, meaning that CL's `read` function will be able to read the data back in. For example, the double-quotes of a string will be included in the output:

```
CL-USER(30): (print "hello")

"hello"
"hello"
```

3.5.3 Princ

`Princ`, as distinct from `print` and `prin1`, prints its output in more of a *human*-readable format, which means for example that the double-quotes of a string will be stripped upon output:

```
CL-USER(31): (princ "hello")
hello
"hello"
```

in the above example, as before, we see the output twice: once when it is actually printed to the console, and again when the return-value of the call to `princ` is printed by the *read-eval-print* loop.

3.5.4 Format

`Format` is a powerful generalization of `prin1`, `princ`, and other basic printing functions, and can be used for almost all output. Format takes a Stream, a *control-string*, and optionally any number of additional arguments to fill in placeholders in the control-string:

```
CL-USER(33): (format t "Hello There ~a" 'bob)
Hello There BOB
NIL
```

In the above example, `t` is used as a shorthand way of specifying `*standard-output*` as the stream, and `~a` is a control-string placeholder which processes its argument as if by `princ`. If you specify either an actual stream or `T` as the second argument to `format`, it will print by side-effect and return `NIL` as its return-value. However, if you specify `NIL` as the second argument, `format` does not output to any stream at all by side-effect, but instead *returns* a string containing what *would* have been output to a stream if one had been provided:

```
CL-USER(34): (format nil "Hello There ~a" 'bob)
"Hello There BOB"
```

In addition to `~a`, there is a wealth of other format directives for an extensive choice of output, such as outputting items as if by `prin1`, outputting lists with a certain character after all but the last item, outputting numbers in many different formats including Roman numerals and English words, etc.

3.5.5 Pathnames

In order to name directories and files on a computer filesystem in an operating-system independent manner, CL provides the *pathname* system. CL pathnames do not contain system-specific pathname strings such as the forward-slash of Unix/Linux filenames or the backward-slash of MSDOS/Windows filenames. A CL pathname is basically a structure which supports a constructor function, `make-pathname`, and several accessor functions, such as `pathname-name`, `pathname-directory`, and `pathname-type`. The pathname structure contains six slots:

- Directory
- Name
- Type
- Device
- Version
- Host

On standard Unix filesystems, only the directory, name, and type slots are relevant. On MSDOS/Windows filesystems, the device slot is also relevant. On the Symbolics platform, which has a more advanced filesystem, the version and host slots also have relevance.

The constructor function, `make-pathname`, takes keyword arguments corresponding to the six possible slots of the pathname.

The following is an example of creating a pathname which would appear on Unix as `/tmp/try.txt`:

```
CL-USER(15): (defparameter *my-path*
                 (make-pathname :directory (list :absolute "tmp")
                                :name "try"
                                :type "txt"))
*MY-PATH*

CL-USER(16): *my-path*
#p"/tmp/readme.txt"
```

As seen in the above example, the `:directory` slot is technically a list, whose `first` is a keyword symbol (either `:absolute` or `:relative`), and whose `rest` is the individual directory components represented as strings.

3.5.6 File Input and Output

Doing file input and output in CL is essentially a matter of combining the concepts of streams and pathnames. You must first *open* a file, which entails associating a stream with the file. CL does provide the `open` function, which directly opens a file and associates a stream to it. For a number of reasons, however, generally you will use a macro

called `with-open-file` which not only opens the file, but automatically closes it when you are done with it, and performs additional cleanup and error-handling details for you. `With-open-file` takes a *specification-list* and a *body*. The specification-list consists of a *stream-variable* (a symbol) and a pathname, followed by several optional keyword arguments which specify, for example, whether the file is for input, output, or both. Here is an example of opening a file and reading one item from it (note that the file denoted by `*mypath*` must exist, or an error will be signalled. You can use the function `probe-file` to test for a file's existence.):

```
(with-open-file (input-stream *my-path*)
  (read input-stream))
```

No extra keyword arguments are required in the specification-list in this example, since "read-only" mode is the default for opening a file. In order to write to a file, however, additional keyword arguments must be given:

```
(with-open-file (output-stream *my-path*
                 :direction :output
                 :if-exists :supersede
                 :if-does-not-exist :create)
  (format output-stream  "Hello There"))
```

3.6 Hash Tables, Arrays, Structures, and Classes

CL provides several other built-in data structures to meet a wide variety of needs. Arrays and hash tables support the storage of values in a manner similar to lists and plists, but with much faster look-up speed for large data sets. Structures provide another plist-like construct, but more efficient and structured.

Classes extend the idea of structures to provide a full object-oriented programming paradigm supporting multiple inheritance, and *methods* which can dispatch based on any number of specialized arguments ("multimethods"). This collection of features in CL is called the Common Lisp Object System, or CLOS.

3.6.1 Hash Tables

A hash table is similar to what is known in some other languages as an associative array. It is comparable to a single-dimensional array which supports keys which are any arbitrary CL object, which will match using `eql`, e.g. symbols or integers. Hash tables support virtually *constant-time* lookup of data, which means that the time it takes to look up an item in a hash table will remain stable, even as the number of entries in the hash table increases.

To work with hash tables, CL provides the constructor function `make-hash-table` and the accessor function `gethash`. In order to set entries in a hash table, use the `setf` operator in conjunction with the `gethash` accessor function. `Setf` is a generalization of `setq` which can be used with *places* resulting from a function call, in addition to exclusively with symbols as with *setq*. Here is an example of creating a hash table, putting a value in it, and retrieving the value from it:

```
CL-USER(35):  (setq os-ratings (make-hash-table))
#<EQL hash-table with 0 entries @ #x209cf822>

CL-USER(36): (setf (gethash :windows os-ratings) :no-comment)
:NO-COMMENT

CL-USER(37): (gethash :windows os-ratings)
:NO-COMMENT
T
```

Note that the `gethash` function returns *two* values. This is the first example we have seen of a function which returns more than one value. The first return-value is the value corresponding to the found entry in the hash table, if it exists. The second return-value is a Boolean value (i.e. `T` or `NIL`), indicating whether the value was actually found in the hash table. If the value is not found, the first return-value will be `NIL` and the second return-value will *also* be `NIL`. This second value is necessary to distinguish from cases when the entry *is* found in the hash table, but its value just happens to be `NIL`.

There are several ways to access multiple return-values from a function, e.g. by using `multiple-value-bind` and `multiple-value-list`.

3.6.2 Arrays

Arrays are data structures which can hold single- or multi-dimensional "grids" full of values, which are indexed by one or more integers. Like hash tables, arrays support very fast access of the data based on the integer indices. You create arrays using the constructor function `make-array`, and refer to elements of the Array using the accessor function `aref`. As with hash tables, you can set individual items in the array using `setf`:

```
CL-USER(38): (setq my-array (make-array (list 3 3)))
#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))

CL-USER(39): (setf (aref my-array 0 0) "Dave")
"Dave"

CL-USER(40): (setf (aref my-array 0 1) "John")
"John"

CL-USER(41): (aref my-array 0 0)
"Dave"

CL-USER(42): (aref my-array 0 1)
"John"
```

The `make-array` function also supports a myriad of optional keyword arguments for initializing the array at the time of creation.

3.6.3 Structures

Structures support creating your own data structures with named slots, similar to, for example, the pathname data structure which we have seen. Structures are defined with the macro `defstruct`, which supports many options for setting up the instance constructor functions, slots, and accessor functions of the structure. For a complete treatment of structures, we refer you to one of the CL references or to the on-line documentation for `defstruct`.

3.6.4 Classes and Methods

Classes and methods form the core of the Common Lisp Object System (CLOS), and add full-blown object-oriented capabilities to Common Lisp.

A complete treatment of CLOS is beyond the scope of this book, but we will provide a brief overview. *Classes* are basically "blueprints" or "prototypes" for objects. *objects*, in this context, are a kind of structure for grouping together both data and computational functionality. *Methods* are very similar to functions, but they can be *specialized* to apply to particular combinations of object types.

CLOS is a *generic function* object-oriented system, which means that methods exist independently from classes (i.e. methods do not "belong to" classes). Methods can take different combinations of class instance types as arguments, however, and can be specialized based upon the particular combinations of types of their arguments. Simple CL datatypes, such as strings and numbers, are also considered as classes.

Here is an example of defining a class and a method, making an instance of the class, and calling the method using the instance as the argument:

```
CL-USER(43): (defclass  box () ((length :initform 10)
                                (width :initform 10)
                                (height :initform 10)))
#<STANDARD-CLASS BOX>

CL-USER(44): (defmethod volume ((self box))
               (* (slot-value self 'length)
                  (slot-value self 'width)
                  (slot-value self 'height)))
#<STANDARD-METHOD VOLUME (BOX)>

CL-USER(45): (setq mybox (make-instance 'box))
#<BOX @ #x209d135a>

CL-USER(46): (volume mybox)
1000
```

As seen in the above example, the class definition takes at least three arguments: a symbol identifying the name of the class, a mixin-list which names superclasses (none, in this example), and a list of slots for the class, with default values specified by the `:initform` keyword argument.

The method definition is very much like a function definition, but its arguments (single argument in the case of this example) are specialized to a particular class type. When we call the method, we pass an instance of the class as an argument, using a *normal argument list*. This is unlike Java or C++, where a method is considered to be "of" a particular class instance, and only additional arguments are passed in an argument list.

CL is more consistent in this regard.

Because normal CL data types are also considered to be classes, we can define methods which specialize on them as well:

```
CL-USER(47): (defmethod add ((value1 string) (value2 string))
               (concatenate 'string value1 value2))
#<STANDARD-METHOD ADD (STRING STRING)>

CL-USER(48): (defmethod add ((value1 number) (value2 number))
               (+ value1 value2))
#<STANDARD-METHOD ADD (NUMBER NUMBER)>

CL-USER(49): (add "hello " "there")
"hello there"

CL-USER(50): (add 3 4)
7
```

3.7 Packages

Typically, symbols in CL exist within a particular *package*. You can think of packages as "area codes" for symbols. They are namespaces used within CL to help avoid name clashes.

Package names are generally represented by keyword symbols (i.e. symbols whose names are preceded by a colon). ANSI CL has a "flat" package system, which means that packages do not "contain" other packages in a hierarchical fashion, but you can achieve essentially the same effect by "layering" packages. Actually, hierarchical packages have been added to Allegro CL as an extension to the ANSI Standard, and may be added to the ANSI standard at some point. However, the real use of hierarchical packages is to avoid name clashes with package names themselves. You can do this in any case simply by separating the components of a package name with a delimiter, for example, a dot: `:com.genworks.books`.

In a given CL session, CL always has a notion of the *current package* which is stored in the (dynamic) variable `*package*`. It is important always to be aware of what the current package is, because if you are in a different package than you think you are, you can get very surprising results. For example, functions that are defined in `:package-1` will not necessarily be visible in `:package-2`, so if you attempt to run those functions from `:package-2`, CL will think they are undefined.

Normally the CL command prompt prints the name of the current package. You can move to a different package, e.g. `foo`, with the toplevel comand `:package :foo`.

Small programs can be developed in the `:user` package, but in general, a large application should be developed in its own package.

3.7.1 Importing and Exporting Symbols

Packages can *export* symbols to make them accessible from other packages, and *import* symbols to make them appear to be local to the package.

If a symbol is exported from one package but not imported into the current package, it is still valid to refer to it. In order to refer to symbols in other packages, qualify the symbol with the package name and a single colon: `package-2:foo`.

If a symbol is not exported from the outside package, you can *still* refer to it, but this would represent a style violation, as signified by the fact that you have to use a double-colon: `package-2::bar`.

3.7.2 The Keyword Package

We have seen many occurences of symbols whose names are preceded by a colon (`:`). These symbols actually reside within the `:keyword` package, and the colon is just a shorthand way of writing and printing them. Keyword symbols are generally used for enumerated values and to name (keyword) function arguments. They are "immune" to package (i.e. there is no possibility of confusion about which package they are in), which makes them especially convenient for these purposes.

3.8 Common Stumbling Blocks

This section discusses common errors and stumbling blocks that new CL users often encounter.

3.8.1 Quotes

One common pitfall is not understanding *when to quote* expressions. Quoting a single symbol simply returns the symbol, whereas without the quotes, the symbol is treated as a variable:

```
CL-USER(6): (setq x 1)
1

CL-USER(7): x
1

CL-USER(8): 'x
X
```

Quoting a list returns the list, whereas without the quotes, the list is treated as a function (or macro) call:

```
CL-USER(10): '(+ 1 2)
(+ 1 2)
```

```
CL-USER(11): (first '(+ 1 2))
+

CL-USER(12): (+ 1 2)
3

CL-USER(13): (first (+ 1 2))
Error: Attempt to take the car of 3 which is not listp.
```

3.8.2 Function Argument Lists

When you *define* a function with `defun`, the arguments go inside their own list:

```
CL-USER(19): (defun square (num)
               (* num num))
SQUARE
```

But when you *call* the function, the argument list comes spliced in directly after the function name:

```
CL-USER(20): (square 4)
16
```

A common mistake is to put the argument list into its own list when calling the function, like this:

```
CL-USER(21) (square (4))
Error: Funcall of 4 which is a non-function.
```

If you are used to programming in Perl or Java, you will likely do this occasionally while you are getting used to CL syntax.

3.8.3 Symbols vs. Strings

Another point of confusion is the difference between symbols and strings. The symbol `AAA` and the string `"AAA"` are treated in completely different ways by CL. Symbols reside in packages; strings do not. Moreover, all references to a given symbol in a given package refer to the same actual address in memory — a given symbol in a given package is only allocated once. In contrast, CL might allocate new memory whenever the user defines a string, so multiple copies of the same string of characters could occur multiple times in memory. Consider this example:

```
CL-USER(17): (setq a1 'aaa)
AAA

CL-USER(18): (setq a2 'aaa)
AAA
```

```
CL-USER(19): (eql a1 a2)
T
```

EQL compares actual pointers or integers — `A1` and `A2` both point to the same symbol in the same part of memory.

```
CL-USER(20): (setq b1 "bbb")
"bbb"

CL-USER(21): (setq b2 "bbb")
"bbb"

CL-USER(22): (eql b1 b2)
NIL

CL-USER(23): (string-equal b1 b2)
T
```

Again, `EQL` compares pointers, but here `B1` and `B2` point to different memory addresses, although those addresses happen to contain the same string. Therefore the comparison by `string-equal`, which compares strings character-by-character, rather than the pointers to those strings, returns `T`.

Another difference between symbols and strings is that CL provides a number of operations for manipulating strings that do not work for symbols:

```
CL-USER(25): (setq c "Jane dates only Lisp programmers")
"Jane dates only Lisp programmers"

CL-USER(26): (char c 3)
#\e

CL-USER(27): (char 'xyz 1)
Error: `XYZ' is not of the expected type `STRING'

CL-USER(29): (subseq c 0 4)
"jane"

CL-USER(30): (subseq c (position #\space c))
" dates only lisp programmers"

CL-USER(31): (subseq c 11 (length c))
"only lisp programmers"
```

```
(defun nearly-equal? (num1 num2 &optional (tolerance *zero-epsilon*))
  (< (abs (- num1 num2)) tolerance))
```

Figure 3.2: Function for Floating-point Comparison

3.8.4 Equality

CL has several different predicates for testing equality. This is demanded because of the many different object types in CL, which dictate different notions of equality.

A simple rule of thumb is:

- Use `eql` to compare symbols, pointers (such as pointers to lists), and integers
- Use `=` to compare most numeric values
- Use `string-equal` for case-insensitive string comparison
- Use `string=` for case-sensitive string comparison
- Use `equal` to compare other types of objects (such as lists and single characters)
- For comparing floating-point numbers which may differ by a very small margin, but should still be considered equal, the safest technique is to take the absolute value of their difference, and compare this with an "epsilon" value within a tolerance of zero. In Figure 3.2 is a simple function which does this, which assumes that a global parameter is defined, `*zero-epsilon*`, as a numeric value very close to zero:

Examples:

```
CL-USER(33): (setq x :foo)
:FOO

CL-USER(34): (eql x :foo)
T

CL-USER(35): (= 100 (* 20 5))
T

CL-USER(36): (string-equal "xyz" "XYZ")
T

CL-USER(37): (setq abc '(a b c))
(A B C)

CL-USER(38): (equal abc (cons 'a '(b c)))
T
```

```
CL-USER(39): (equal #\a (char "abc" 0))
T

CL-USER(40): (eql abc (cons 'a '(b c)))
NIL
```

The last expression returns NIL because, although the two lists have the same contents, they reside in different places in memory, and therefore the pointers to those lists are different.

Note that several of the functions which by default use `eql` for matching will take an optional keyword argument `test` which can override this default:

```
CL-USER(54): (member "blue" '("red" "blue" "green" "yellow")
                    :test #'string-equal)
("blue" "green" "yellow")
```

Recall from Section 3.2.7 that `member` returns the `rest` of the list starting at the found element.

3.8.5 Distinguishing Macros from Functions

Since macros[4] and functions both occur as the `first` element in an expression, at first glance it may appear difficult to distinguish them. In practice, this rarely becomes an issue. For beginners in the language, the best approach is to make a default assumption that something is a function unless you recognize it as a macro. Most macros will be readily identifiable because one of the following will apply:

- It is a familiar core part of CL, such as `if`, `cond`, `let`, `setq`, `setf`, etc.
- Its name begins with "`def`," such as with `defun`, `defparameter`, etc.
- Its name begins with "`with-`," as in `with-open-file`, `with-output-to-string`, etc.
- Its name begins with "`do`," for example `dolist` and `dotimes`.

Beyond these general rules of thumb, if you find yourself in doubt about whether something is a function or a macro, you can:

1. Check with on-line reference documentation or in a CL reference book (see the Bibliography in Appendix B)
2. Use the `symbol-function` function to ask your CL session, like this:

```
CL-USER(24): (symbol-function 'list)
#<Function LIST>

CL-USER(25): (symbol-function 'with-open-file)
#<macro WITH-OPEN-FILE @ #x2000b07a>
```

[4] *Special Operators* are one other category of operator in CL, which internally are implemented differently from macros, but which for our purposes we can consider like macros.

3.8.6 Operations that cons

Because CL programs do not have to free and allocate memory explicitly, they can be written much more quickly. As an application evolves, some operations which "cons" can be replaced by operations which "recycle" memory, reducing the amount of work which the garbage collector (CL's automatic memory management subsystem) has to do.

In order to write programs which conserve on garbage collecting, one technique is simply to avoid the unnecessary use of operations which cons. This ability will come with experience.

Another technique is to use `destructive` operations, which are best used only after one has gained more experience working with CL. Destructive operations are for the most part beyond the scope of this book.

Chapter 4

Interfaces

One of the strong points of modern commercially-supported CL environments is their flexibility in working with other environments. In this chapter we present a sampling of some interfaces and packages which come in handy for developing real-world CL applications and connecting them to the outside environment. Some of the packages covered in this chapter are commercial extensions to Common Lisp as provided with Franz Inc's Allegro CL, and some of them are available as open-source offerings.

4.1 Interfacing with the Operating System

One of the most basic means of interacting with the outside world is simply to invoke commands at the operating system level, in similar fashion to Perl's "system" command. In Allegro CL, one way to accomplish this is with the function `excl.osi:command-output`[1]. This command takes a string which represents a command (possibly with arguments) to be executed through an operating system shell, such as the "C" shell on Unix:

```
(excl.osi:command-output "ls")
```

When invoked in this fashion, this function returns the the Standard Output from the shell command as a string. Alternatively, you can specify that the output go to another location, such as a file, or a variable defined inside the CL session.

When using `excl.osi:command-output`, you can also specify whether CL should wait for the shell command to complete, or should simply go about its business and let the shell command complete asynchronously.

4.2 Foreign Function Interface

Allegro CL's *foreign function interface* allows you to load compiled libraries and object code from C, C++, Fortran, and other languages, and call functions defined in this code

[1] You must have a fully patched ACL 6.2 or later ACL version to access the excl.osi functionality (if your machine can access the internet, you can bring your ACL installation to the current patch level with the function `sys:update-allegro`). Previous ACL versions have a function `excl:run-shell-command` which operates similarly to `excl.osi:command-output`.

as if they were defined natively in CL. This technique requires a bit more setup work than using simple shell commands with `excl.osi:command-output`, but it will provide better performance and more transparent operation once it is set up.

4.3 Interfacing with Corba

Corba, or the Common Object Request Broker Architecture, is an industry-standard mechanism for connecting applications written in different object-oriented languages.

To use Corba, you define a standard interface for your application in a language called Interface Definition Language (Corba IDL), and each application must compile the standard interface and implement either a servant or a client for the interface. In this manner, servants and clients can communicate through a common protocol, regardless of what language they are written in.

In order to use Corba with CL requires a Corba IDL compiler implemented in CL, as well as a run-time ORB, or Object Request Broker, to handle the actual client/server communications. Several such Corba IDL compilers and ORBs are available for CL. An example of such a system is Orblink, which is available as an optional package for Allegro CL. Orblink consists of a complete Corba IDL compiler, and a CL-based ORB, which runs as a CL thread rather than as a separate operating system process.

Using Orblink to integrate a CL application to a Corba client/server environment is a straightforward process. To start, you simply obtain the Corba IDL files for the desired interface, and compile them in CL with the command `corba:idl`. This will automatically define a set of classes in CL corresponding to all the classes, methods, etc. which make up the interface.

To use CL as a client, you can now simply make instances of the desired classes, and use them to call the desired methods. The Orblink ORB will take care of communicating these method calls across the network, where they will be invoked in the actual process where the corresponding servant is implemented (which could be on a completely different machine written in a completely different language).

To use CL as a servant, you must implement the relevant interfaces. This consists of defining classes in CL which *inherit* from the "stub" servant classes automatically defined by the compilation of the Corba IDL, then defining methods which operate on these classes, and which perform as advertised in the interface. Typically, a Corba interface will only *expose* a small piece of an entire application, so implementing the servant classes would usually represent a small chore relative to the application itself.

4.4 Custom Socket Connections

Just about every CL implementation provides a mechanism to program directly with network sockets, in order to implement "listeners" and network client applications. As a practical example, Figure 4.1 shows a Telnet server, written by John Foderaro of Franz Inc., in 22 lines of code in Allegro CL. This Telnet server will allow a running CL process to accept Telnet logins on port 4000 on the machine on which it is running, and will present

```
(defun start-telnet (&optional (port 4000))
  (let ((passive (socket:make-socket :connect :passive
                                     :local-host "127.1"
                                     :local-port port
                                     :reuse-address t)))
    (mp:process-run-function
     "telnet-listener"
     #'(lambda (pass)
         (let ((count 0))
           (loop
             (let ((con (socket:accept-connection pass)))
               (mp:process-run-function
                (format nil "tel~d" (incf count))
                #'(lambda (con)
                    (unwind-protect
                        (tpl::start-interactive-top-level
                         con
                         #'tpl::top-level-read-eval-print-loop
                         nil)
                      (ignore-errors (close con :abort t))))
                con)))))
     passive)))
```

Figure 4.1: Telnet Server in 22 Lines

the client with a normal CL Command Prompt (for security, it will by default only accept connections from the local machine).

4.5 Interfacing with Windows (COM, DLL, DDE)

Most Commercial CL implementations for the Microsoft Windows platform will allow CL applications to use various Microsoft-specific means for integrating with the Microsoft platform. For example, an Allegro CL application can be set up to run as a COM server, or compiled into a DLL to be used by other applications.

4.6 Code Generation into Other Languages

CL excels at reading and generating CL code. It is also a powerful tool for analyzing and generating code in other languages. An example of this can be found in Franz Inc's htmlgen product, which provides a convenient CL macro to generate HTML for a web page.

Another example is MSC's AutoSim product, which uses a CL program to model an automobile, then generates optimized C programs for solving specialized systems of linear equations relating to vehicle dynamics.

CL also played a crucial role in preventing the potential Y2K disaster by automatically analyzing and repairing millions of lines of COBOL!

4.7 Multiprocessing

Multiprocessing allows your application to carry on separate tasks virtually simultaneously. Virtually, because on a single-processor machine, only one operation can physically be happening at any one time. However, there are situations when you want your program to *appear* to be executing separate execution threads, or processes, simultaneously. Web serving is a good example. As described in Section 4.9, you may be running your CL application as a webserver. This means that multiple users, on different machines in different parts of your company or even around the world, may be sending requests to your application in the form of web browser (client) HTTP commands. If one user's request takes an especially long time to process, you probably do not want all your other users to have to wait for a response while that one request is being processed.

Multiprocessing addresses such problems by allowing your CL application to continue servicing other requests, while "simultaneously" completing that long computation.

Allegro Multiprocessing is covered in the document `multiprocessing.htm` which ships with all editions of Allegro CL.

4.7.1 Starting a Background Process

Starting another thread of execution in Allegro CL can be as simple as calling the function `mp:process-run-function`. This function spawns a separate thread of execution, while the execution of code from whence you call it continues. Here is a simple example which you can run from the command line:

```
(defvar *count* 0)
(mp:process-run-function
 (list :name "counting to a million in background"
       :priority -10)
  #'(lambda() (dotimes (n 1000000) (setq *count* n))))
```

This starts a thread named "counting to a million in background" with a relative priority of -10. Notice that after you call this function, your toplevel prompt returns immediately. The toplevel Read-Eval-Print loop is continuing on its merry way, while your new process is running in the background, counting to a million (actually, 999999). While the background process is running, you can sample the value of `count` at any time:

```
CL-USER(16): *count*
424120
CL-USER(17): *count*
593783
CL-USER(18): *count*
679401
CL-USER(19): *count*
765012
```

Note that on a fast processor, you may need to increase the count value in this example to be able to take samples before the code completes.

4.7.2 Concurrency Control

Sometimes you want to prevent two threads from modifying a value in memory at the same time. For this purpose, you can use *process locks*. A process lock is an object which can be *seized* by at most one running process at a time. Using process locks, you can ensure that multiple threads of execution will interact as you expect.

In our previous counting example, let's say you want to prevent another thread from modifying the `count` variable until the count is complete. First, you would establish a process lock specific for this purpose:

```
CL-USER(20): (setq count-lock (mp:make-process-lock))
#<MULTIPROCESSING:PROCESS-LOCK  @ #x5a02fba>
```

Now, run the background function within the context of seizing this lock:

```
CL-USER(21): (mp:process-run-function
               (list :name "counting to a million in background"
                     :priority -10)
               #'(lambda()
                   (mp:with-process-lock (count-lock)
                     (dotimes (n 1000000) (setq count n)))))
#<MULTIPROCESSING:PROCESS counting to a million in background @ #x5a0332a>
```

You now can adopt a policy that any piece of code wishing to modify the value of `count` must first seize the `count-lock`. This will ensure that nothing will interfere with the counting process. If you do the following while the counting process is running:

```
CL-USER(22): (mp:with-process-lock (count-lock) (setq count 10))
```

you will notice that execution will *block* until the counting process is finished. This is because the `count-lock` is already seized, and execution is waiting for this lock to be released before proceeding.

It is usually a good idea to make your process locks as *fine-grained* as possible, to avoid making processes wait unecessarily.

From your toplevel prompt, you can monitor which processes are running in the current CL image with the toplevel `:proc` command:

```
CL-USER(28): :proc
P Bix Dis   Sec  dSec  Priority  State   Process Name, Whostate, Arrest
*   7   1     2   2.0       -10 runnable counting to a million in background
*   1   8    28   0.1         0 runnable Initial Lisp Listener
*   3   0     0   0.0         0 waiting  Connect to Emacs daemon, waiting for input
*   4   0     0   0.0         0 inactive Run Bar Process
*   6   0     3   0.0         0 waiting  Editor Server, waiting for input
```

See the `multiprocessing.htm` for details on each column in this report, but essentially this is showing the state of each process currently in the system. This example shows that Allegro CL uses separate threads for maintaining the connection to the Emacs text editor.

As we will see in Section 4.9, the AllegroServe webserver automatically creates and manages separate threads to deal with simultaneous web connections.

4.8 Database Interfaces

CL in general, and Allegro CL specifically, provides a myriad of mechanisms for connecting to databases. Most of these mechanisms differ only slightly in their operation, so you can generally switch among the different mechanisms without major rework in your application. The two mechanisms we will cover here are both included with the Enterprise and Platinum editions of Allegro CL.

4.8.1 ODBC

ODBC, or Open Database Connectivity, is a standard means of connecting to a relational database based on SQL (Structured Query Language, the industry standard Data Definition and Data Manipulation language for relational databases). Virtually all popular database systems can be connected to using ODBC. On Windows these facilities are usually free of charge; on Unix they may or may not be free. In any case, ODBC is a good solution to consider if you want to connect your application to several different relational database systems without writing and maintaining separate pieces of code specific to each.

ODBC uses the concept of *data sources*, each with a name. There are operating system dependent ways to set up data sources on your machine to point to actual physical databases, which may be housed on the local machine or remote. Assuming we have a data source named "`login`" with a table named "`USER`," connecting to and querying the database can be as simple as the following:

```
CL-USER(40): (setq handle (dbi:connect :data-source-name "login"))
#<DBI::ODBC-DB "DSN=login;DB=login;...[truncated]" @ #x4b2c3b2>
CL-USER(41): (dbi:sql "select * from USER" :db handle)
(("1" "dcooper8" "guessme" "Dave" NIL "Cooper" NIL NIL NIL NIL ...)
 ("7" "awolven" NIL NIL NIL NIL NIL NIL NIL NIL ...))
("GEN_ID" "LOGIN" "PASSWORD" "FIRST_NAME" "MIDDLE_NAME" "LAST_NAME"
 "COMPANY" "ADDRESS_1" "ADDRESS_2" "CITY" ...)
```

As with most CL database interfaces, the ODBC interface returns all SQL queries with multiple values: The first (and main) return value is a list of lists, corresponding to the rows in the returned query. The second return value is a list of field names, corresponding to the values in each of the first lists.

Because ODBC by definition must retain portability among database systems, it is generally unable to take full advantage of database-specific features and performance optimizations.

4.8.2 MySQL

MySQL is a popular relational database which is available both in commercially supported and free, open-source form. While you can connect to a MySQL database using ODBC, Allegro CL also provides a direct connection to MySQL, which performs faster and is easier to set up than the ODBC connection. From the application level, its use is simple enough that an application using the MySQL Direct interface could be converted to use ODBC without major rework, should that become necessary. Therefore, if you know your Allegro CL application will be using MySQL for the immediate future, we recommend using the MySQL Direct interface rather than ODBC.

Here is the same example from above, using the MySQL Direct interface instead of the ODBC one:

```
CL-USER(50): (setq handle (dbi.mysql:connect  :user "dcooper8"
                                              :password "guessme"
```

```
                                                  :database "login"))
#<DBI.MYSQL:MYSQL connected to localhost/3306 @ #x5bd114a>
CL-USER(51): (dbi.mysql:sql "select * from USER" :db handle)
((1 "dcooper8" "bhaga<<" "Dave" #:|null| "Cooper" #:|null| ...)
 (7 "awolven" #:|null| #:|null| #:|null| #:|null| #:|null| ...))
("GEN_ID" "LOGIN" "PASSWORD" "FIRST_NAME" "MIDDLE_NAME"
 "LAST_NAME" "COMPANY" "ADDRESS_1" "ADDRESS_2" "CITY" ...)
```

A few differences from ODBC to note:

- The functions are in the `dbi.mysql` package rather than the `dbi` package.
- Since there is no need to set up separate data sources for MySQL Direct, we connect to a MySQL instance on the local machine directly by specifying a user, password, and database name.
- NULL values in the return query are represented with the special symbol `#:|null|` rather than with `NIL` as in ODBC.

4.9 World Wide Web

4.9.1 Server and HTML Generation

The dynamic and multi-threaded nature of AllegroCL makes it a natural fit for powering webserver applications. Franz provides a popular open-source webserver called AllegroServe which is designed just for this purpose.

With a standard Allegro CL 6.2 installation you can load AllegroServe and access its exported symbols as follows:

```
(require :aserve)
(use-package :net.aserve)
(use-package :net.html.generator)
(use-package :net.aserve.client)
```

With AllegroServe, you can map website addresses to actual functions which will run in response to a request. Here is a simple example:

```
(defun hello (req ent)
  (with-http-response (req ent)
    (with-http-body (req ent)
      (html "Hello World"))))

(net.aserve:publish :path "/hello.html"
                    :function #'hello)
```

Now, whenever a web visitor goes to the `/hello.html` URI on our server, she will receive in response a page with the simple text "Hello World." More dynamic examples would of course be more interesting, and we present some in the next chapter.

Note the use of the `html` operator in the previous example. This is a macro from the HTMLgen package which ships with Allegroserve. HTMLgen allows convenient generation of dynamic, well-structured HTML from a lisp-like source format. Please see the AllegroServe and HTMLgen documentation at `http://opensource.franz.com/aserve/index.html` for details on both of these.

4.9.2 Client and HTML Parsing

Where there is a server there must be a client. In the World Wide Web, we typically think of clients as being interactive web browsers such as Netscape or Internet Explorer. However, web clients can also be autonomous programs. This is how "agents" and "crawlers" work – they are actually programmatic web browsers which continually send requests to webservers and parse the results, gathering information.

Allegroserve includes a web client for such purposes. Using it can be as simple as a call to `net.aserve.client:do-http-request`. For example, to retrieve the URI from the above example programmatically:

```
CL-USER(1): (do-http-request "localhost:9000/hello.html")
"Hello World"
200
((:TRANSFER-ENCODING . "chunked") (:SERVER . "AllegroServe/1.2.25")
 (:DATE . "Tue, 04 Mar 2003 21:36:20 GMT") (:CONNECTION . "Close"))
#<URI http://localhost:9000/hello.html>
```

As you can see, the `do-http-request` function returns four values:

1. the actual contents of the response, as a string
2. the response code (e.g. 200 for a normal response, 404 for a "Page Not Found," etc.)
3. an assoc-list containing the HTTP headers
4. a URI object, an internal CL object containing information about the URI

For more complex client requests which return structured HTML, an HTML parser is available. For example, you might create a news agent by sending an HTTP request to a news site, then parsing the HTML to create convenient list structures from which you can elicit the text of the news stories. Please see `http://opensource.franz.com/xmlutils/index.html` for details on this parser.

4.10 Regular Expressions

Many applications have to work with strings of characters. Doing this in an efficient manner can be a key to the efficiency and performance of your entire application. Allegro CL provides a Regular Expression module for this purpose. The Regular Expression module allows you to search, replace, and manipulate patterns in character strings in an efficient and optimized manner.

As a simple example, let us replace one character in a string with another. Assume we have some field names as variables in CL, which can legally contain dash ("-") characters. Say we want to use these names as a basis for field names in an SQL database, where dashes are not allowed as part of a name. We will replace all dashes with underscores:

```
(defun replace-dashes (word)
  (excl:replace-regexp word "-" "_"))

CL-USER(59): (replace-dashes "top-o-the-morning")
"top_o_the_morning"
```

Please see the document `regexp.htm`, which ships with all editions of Allegro CL, for a full Regular Expression reference.

4.11 Email

Email was the first "killer app" of the Internet, and arguably continues in that role to this day. Despite threats from viruses and unsolicited commercial email, many people in the modern world still depend on email for their daily communication. CL provides an excellent environment for automating email processing. Not coincidentally, in the next chapter we present an example of using CL to perform text classification on email, an effective means for filtering unwanted and unsolicited commercial bulk email.

4.11.1 Sending Mail

Franz provides a convenient open-source package for sending messages to an SMTP (Simple Mail Transfer Protocol) server.

With a standard Allegro CL 6.2 installation you can load this package and access its exported symbols as follows:

```
(require :smtp)
(use-package :net.post-office)
```

This package provides the `send-letter` function. This function takes the name of an SMTP server along with the recipient, the text of an email, and other optional arguments, and actually delivers the mail to the specified SMTP host. Here is an example:

```
(send-letter "smtp.myisp.com" "dcooper@genworks.com" "joe@bigcompany.com"
 "See you at the game tonight." :subject "Tonight's Game"
```

That's all it takes to have your application shoot off an email.

4.11.2 Retrieving Mail

The two main Internet standards for retrieving mail are POP and IMAP. Although IMAP is newer and more feature-filled than POP, many ISPs still support only POP, so we will cover the POP interface here. Franz provides a single file, `imap.cl` which contains client interfaces both for POP and IMAP.

With a standard Allegro CL 6.2 installation you can load this package and access its exported symbols as follows:

```
(require :imap)
(use-package :net.post-office)
```

Here is an example of using the POP client interface. First, you must create a connection handle to the POP server:

```
CL-USER(4): (setq mailbox (net.post-office:make-pop-connection "pop.myisp.com"
                                                               :user "joe"
                                                               :password "guessme"))
#<NET.POST-OFFICE::POP-MAILBOX @ #x5dcc33a>
```

Now we can get a count of messages in the Inbox:

```
GDL-USER(5): (net.post-office:mailbox-message-count mailbox)
1
```

and fetch the one message:

```
GDL-USER(9): (net.post-office:fetch-letter mailbox 1)
"Content-type: text/plain; charset=US-ASCII
MIME-Version: 1.0 ...
..."
```

Finally, we can delete this message and close our connection:

```
GDL-USER(10): (net.post-office:delete-letter mailbox 1)
NIL
GDL-USER(12): (net.post-office:close-connection mailbox)
T
```

Please see the documentation at `http://opensource.franz.com/postoffice/index.html` for complete coverage of the POP and IMAP facilities.

In the next chapter, we will combine most of what we have learned in this chapter to put together a web-based program for browsing and classifying email. Because we make use of pre-built libraries and interfaces such as the ones described in this chapter, we can create a functioning multi-user application with less than 500 lines of code.

Chapter 5

Squeakymail

This chapter will present a sample application in CL, called "Squeakymail," for filtering and browsing email. Squeakymail demonstrates the use of many of Allegro CL's interfaces and special features covered in the previous chapter, in an application of under 500 lines.

Please note that Squeakymail is an open-source application and is still evolving, so you should obtain the latest code if you wish to work with it. Please contact Genworks at `info@genworks.com` for information on obtaining the current Squeakymail codebase.

5.1 Overview of Squeakymail

Squeakymail is a multi-user, server-based application which performs two main tasks:

1. It fetches mail for each user from one or more remote POP (Post Office Protocol) accounts, analyses it, scores it for its probability of being "spam" (junk email), and stores it in a local queue for final classification by the user. This fetching goes on perpetually, as a background thread running in the CL image.

2. It provides a simple web-based interface for each user to be able to view the incoming mail headers and bodies, and confirm or override the pre-scoring on each message. Once classified, the individual tokens (words) from each message are added to in-memory spam probability tables, so that the scoring accuracy can become better over time.

There are some obvious extensions to Squeakymail which could be done, for example providing an option to auto-classify messages whose spam probability exceeds a certain threshhold, avoiding the need for the user even to see those messages.

5.2 Fetching and Scoring

One part of our application will run on a background thread, periodically polling users' remote POP3 email sources for new messages. Specifically, the mail fetching thread will perform the following for each remote POP3 account of each local user:

```
(defun fetch-mails ()
  (mp:process-run-function "fetching mail in the background"
    #'(lambda()
        (do ()()
          (let ((start (get-universal-time)))
            (with-db-rows ((login) :db *mysql-logins* :table "USER")
              (fetch-mail login))
            (sleep (max 0 (- *mail-fetching-interval*
                             (- (get-universal-time) start)))))))))
```

Figure 5.1: Starting a Background Thread

1. Grab a process lock for the current local user, to prevent other threads from simultaneously modifying that user's data (files or in-memory).
2. Create a POP3 connection to the specified POP3 server.
3. Open the local user's local Incoming Mail file for appending.
4. Fetch the current Message Count (i.e., number of messages) from the POP server
5. Iterate through each numbered message, doing the following:
   (a) Fetch the entire contents of the message into memory.
   (b) Parse the message into its individual header elements and its body.
   (c) Compute individual tokens for each header and for the body.
   (d) Compute a spam probability "score" based on these tokens.
   (e) Write an entry into the local user's Incoming Mail file, containing the headers, body, list of tokens, and spam probability score
   (f) Finally, mark the message for deletion from the remote POP3 server.

Figures 5.1 and 5.2 contain the basic code for this part of the program. The code in Figure 5.1 starts the background thread with a call to `mp:process-run-function`, giving a name to the process and a lambda expression (anonymous function object) to be run by the process. That function enters into an infinite iteration (with `do`). On each iteration of this infinite loop, it scans through the table of local login accounts using the `with-db-rows` macro from the dbi.mysql package. For each row, the lexical variable `login` is set to the respective user's login name, and the `fetch-mail` function from Figure 5.2 is invoked for that user. Finally, the process goes into a Sleep mode until the next scheduled interval, at which point it will wake up and perform yet another iteration of its infinite loop.

The code in Figure 5.2 fetches and scores mail for an individual local user. It first establishes a process lock for that particular user for the context of the rest of the body of the function. The lock object itself comes from a call to the function `user-process-lock` (Figure 5.3), which retrieves from a hash table a lock specific to the local user, or creates a

```
(defun fetch-mail (user)
 (mp:with-process-lock ((user-process-lock user))
   (with-db-rows ((host login password) :table "POP_ACCOUNT"
                                        :db *mysql-local*
                                        :where (=text local_login user))
     (format t "Fetching mail from ~a for local user ~a~%" host user)
     (with-open-pop-box (mb host login password)
       (with-open-file
           (out (string-append (format nil "/var/mail/incoming/~a" user))
            :direction :output :if-exists :append :if-does-not-exist :create)
         (let ((message-count (mailbox-message-count mb)))
           (dotimes (n message-count)
             (let* ((message (fetch-letter mb (1+ n))))
               (multiple-value-bind (headers body)
                   (parse-mail-header message)
                 (let ((tokens (tokenize headers body)))
                   (let ((*package* *token-package*))
                     (multiple-value-bind (score scored-tokens)
                         (score-tokens tokens user)
                       (print (list :headers headers
                                    :body body
                                    :tokens tokens
                                    :score score
                                    :scored-tokens scored-tokens) out)))))))
           (delete-letter mb (1+ n)))))))))
```

Figure 5.2: Fetching and Scoring Remote Mail for a Local User

```
(defun user-process-lock (user)
  (or (gethash user *mailbox-locks-hash*)
      (setf (gethash user *mailbox-locks-hash*)
        (mp:make-process-lock :name (format nil "fetching mail for ~a" user)))))
```

Figure 5.3: Retrieving or Creating a Process Lock

```
(defmacro with-open-pop-box
    ((handle server user password &optional (timeout 10)) &body body)
  `(let ((,handle (make-pop-connection ,server :user ,user :password ,password
                                       :timeout ,timeout)))
     (unwind-protect (progn ,@body)
       (when ,handle (close-connection ,handle)))))
```

Figure 5.4: A macro to open and ensure clean closing of a POP3 connection

new lock if one does not yet exist. Running the rest of the function body within this process lock will ensure that no other threads of execution will interfere with this user's data during the fetching operation. For example, another part of our application will provide a web interface for users to browse and classify their mail interactively. This classification also modifies the user's Incoming Mail file, usually by deleting one or more messages from it. By running our mail fetching inside a user-specific process lock, we eliminate any chance of these different update operations happening simultaneously. Doing so would likely corrupt the Incoming Mail file, resulting in lost messages or worse.

This process locking does mean, however, that interactive users might experience a slight delay when trying to visit their Mail Classification web page, if they happens to do this while a fetch cycle their her mail is in progress. This is because the interactive process also requires that same user-specific lock, and will go into "wait" mode until the fetching process completes and releases the lock.

The `fetch-mail` function next prints a status message to the console indicating the local user and remote host for the current fetching operation.

Next it enters into the code written by the `with-open-pop-box` macro (Figure 5.4), which opens a connection to the specified POP3 account, assigns the name `mb` to this connection, and finally guarantees that the connection will be closed when the code within `with-open-pop-box` either completes or results in any kind of error condition. Now the `fetch-mail` function enters another context, `with-open-file`, which gives us a stream named `out` for writing to the user's Incoming Mail file.

Now we query the `mb` POP connection for the number of messages, and bind this result to the variable `message-count`. This tells us how many times we have to fetch a message, which we do inside the code of a `dolist`, using the `fetch-letter` function from the `net.post-office` package. Note that we add one (1) to the `dotimes` variable `n` when fetching the letter, since POP3 message numbers begin with 1, rather than with 0 as is the case for `dolist` iterator variables.

Now we parse the message into headers and body, using `multiple-value-bind`, since the `parse-mail-header` function returns the headers and body as two return values.

Next we *tokenize* the message (described in Section 5.2.1), compute a score based on the tokens (Section 5.2.2), and write all the relevant information to the user's Incoming Mail file in the convenient form of a plist.

Finally, we mark the message for deletion from the remote POP server, with the `delete-letter` function, also from the `:net.post-office` package.

```
(defun tokenize (headers body)
  (setq body (excl:replace-regexp
              (excl:replace-regexp body "<!-.[^>]*->" "") "<![^>]*>" ""))
  (append (tokenize-headers headers)
          (mapcan #'(lambda(item) (if (stringp item)
                                      (tokenize-string item)
                                    (list item)))
                  (flatten (parse-html body)))))

(defun tokenize-headers (headers)
  (mapcan #'(lambda(cons)
              (let ((header (first cons))
                    (string (rest cons)))
                (let ((strings (excl:split-regexp "\b+" string)))
                  (mapcar #'(lambda(string)
                              (intern (string-append header "*" string)
                                      *token-package*)) strings)))) headers))

(defun tokenize-string (string)
  (let ((words (excl:split-regexp "\b+" string)) result)
    (dolist (word words (nreverse result))
      (when (and (<= (length word) *max-token-length*)
                 (tokenizable? word))
        (push (intern word *token-package*) result)))))
```

Figure 5.5: Tokenizing using Regular Expression facility

5.2.1 Tokenizing

Tokenizing, the process of breaking up a text into individual words or units, is at the heart of our spam-filtering strategy. Our current version of tokenizing makes heavy use of Allegro CL's Regular Expression API, a set of functions for manipulating text. The three functions listed in Figure 5.5 comprise our current tokenization system. The first function, `tokenize`, first uses `excl:replace-regexp` to remove all HTML and XML comments from the body (inserting random HTML comments is a common trick used by spammers to confound filters). The function then returns the list of tokenized headers appended to the list of tokenized words from the message body. To produce the list of tokenized words, we first use the Franz HTML parser to pull out individual HTML tags and strings (this will have no effect if the message does not have HTML to begin with). Since the parser returns a nested list of tags, we use `flatten` to convert it into a flat list. Then we pass each string into the `tokenize-string` function.

`Tokenize-headers` accepts a list of headers, each of which is a dotted list whose first is the name of the header, and whose rest is a string with the value of the header. We split the value of the header into individual words by using `excl:split-regexp` to split on whitespace. For each word thus obtained, we prepend it with the name of the header. This distinguishes header tokens from body tokens, since a given token in a Subject line, for example, can have a very different spam probability from the same token in the body of the message. Thus, a message with:

```
Subject: I Love You
```

will generate tokens `Subject*I`, `Subject*Love`, and `Subject*You`.

The `tokenize-string` function takes a string and essentially splits it on whitespace to produce individual words. We reject words longer than `*max-token-length*`, and check that the word is "tokenizable," meaning that it will not confuse the CL reader when reading the token as a symbol. Currently the only known character sequence which will confuse the CL reader is a sequence of more than one dot, e.g. "..." so these are rejected as well. All the strings are converted to symbols and interned into a certain CL package we have created for this purpose. Converting the strings into symbols makes it much faster to look up the tokens in hash tables later.

5.2.2 Scoring

Once we have a message tokenized, the actual score computation is fairly straightforward. We call the function `score-tokens` with the list of tokens extracted from the message. This function uses Paul Graham's[1] "Plan for Spam" technique to score the messages. It assumes the existence of a hash table mapping tokens to their individual spam probabilities. In Section 5.3 we will see how these hash tables are created and maintained, on a per-user basis.

The scoring function first binds `ht` to the user-specific probabilities table using the function `restore-probabilities`, which retrieves the probability table from memory, restores it from a saved file, or creates a new one if it does not yet exist. Next it looks up each

[1] http://www.paulgraham.com/spam.html

```
(defun score-tokens (tokens user)
  (let ((ht (restore-probabilities user)))
    (let* ((scored-tokens
            (sort (mapcar #'(lambda(token)
                              (list token (or (gethash token ht)
                                              *unknown-probability*)))
                          tokens)
                  #'> :key #'(lambda(token) (abs (- (second token) 0.5)))))
           (probs (mapcar #'second
                          (subseq scored-tokens 0
                                  (min *number-of-relevant-tokens*
                                       (length scored-tokens))))))
      (values
       (let ((prod (apply #'* probs)))
         (/ prod (+ prod (apply #'* (mapcar #'(lambda (x) (- 1 x)) probs)))))
       scored-tokens))))
```

Figure 5.6: Scoring a message based on its tokens

token in the probabilities table, creating a list pairing each token with its corresponding spam probability. It then sorts this result based on the distance of the probability from the neutral 0.5.

The actual probability values, `probs`, is then computed by taking the top `*number-of-relevant-tokens*` (default 15) values from this list. The combined probability is computed using Bayes' formula as described in Paul Graham's Plan for Spam paper.

A possible improvement to the scoring, mentioned by Paul Graham, would be not to allow duplicate tokens (or limit the number of duplicate tokens) in computing the top values.

5.3 Browsing and Classifying

Our application's "left brain" will present an interface to users, allowing them to "train" the mail classifier. The interface covered here is somewhat bare-bones; we have also prepared a more feature-full interface using Genworks' GDL/GWL system. The code for that is included in Appendix A.

The interface presented here is missing some key features. The most obvious one is a security mechanism to prevent someone from accessing another user's mail (GWL provides a built-in mechanism for login and password security; such a system is left as an exercise for those who wish to implement something like Squeakymail in Common Lisp and AllegroServe).

Our interface will present a dynamically created web page which will show the headers from the user's incoming mail, along with the score of each and resulting spam/nonspam status. Using a radio button control for each mail, the user will be able to correct any misclassifications.

```
(defun classify (req ent)
  (let ((query (request-query req)))
    (let ((keys (mapcar #'first query))
          (values (mapcar #'rest query)))
      (let ((user (rest (assoc "user" query :test #'string-equal)))
            (spam-indices (let ((ht (make-hash-table :values nil)))
                            (mapc #'(lambda(key value)
                                      (when (and (search "spamp-" key) (string-equal value "SPAM"))
                                        (setf (gethash (read-from-string (subseq key 6)) ht) t))) keys values) ht))
            (nonspam-indices (let ((ht (make-hash-table :values nil)))
                               (mapc #'(lambda(key value)
                                         (when (and (search "spamp-" key) (string-equal value "NON-SPAM"))
                                           (setf (gethash (read-from-string (subseq key 6)) ht) t))) keys values) ht)))
        (let ((messages (classify-and-write-files user spam-indices nonspam-indices)))
          (mp:process-run-function (list :name (format nil "Squeakymail: Recomputing probabilities for user ~a in the background" user)
                                         :priority -5 :quantum 1) #'(lambda()(sleep 5)(compute-probabilities user)))
          (with-http-response (req ent) (with-http-body (req ent) (classify-form messages user)))))))))

(publish :path "/classify" :function #'classify)
```

Figure 5.7: Main Function to produce Classification Web Page

The main `classify` function is shown in Figure 5.7, and is published (mapped to a URL) also in Figure 5.7. The `classify` function, as with all Allegroserve http response functions, takes `req` and `ent` arguments. The `req` argument comes from the browser, and contains the values from any submitted html forms. These values are bound to the `query` variable. When the `classify` function is called for the first time, it contains no form data. On subsequent invocations, however, it might contain form data indicating which messages are spam and nonspam. Later we will see how to create the form to submit these values.

Once we extract the form keys (field names) and values from the query information, we compute two hash tables: `spam-indices` and `nonspam-indices`. These contain the numerical indices representing the user's selection of radio buttons on each message. Next we call the `classify-and-write-files` function with the `spam-indices` and `nonspam-indices`. The internals of this function will be described soon; basically it moves messages from the user's incoming mail file, and returns any remaining (unclassified) messages along with any new messages which have arrived in the user's incoming mail file in the meantime.

Then we create a temporary file as `incoming-buffer-pathname` to contain the unclassified messages, `spams-pathname` to contain our spam corpus, `nonspams-pathname` to contain our nonspam corpus, and `popqueue` for appending to the user's actual mail file.

Now, using a `cond` expression, we distribute each message to the appropriate file.

The `classify` function then spawns a low-priority background process to recompute the spam token probabilities for the user.

Finally, it generates the actual web page with messages and classification form, inside a `with-http-response` and `with-http-body`, by calling the `classify-form` function.

Figure 5.8 contains the code for the `classify-and-write-files` function. This function takes a list of spam-indices and nonspam-indices, and moves messages around appropriately. This is all done within the context of the user-specific process lock, to prevent two user requests from rearranging files simultaneously which might result in corrupted files.

The `classify-and-write-files` function first populates a list of messages, bound to the `let` variable `messages` by opening and reading from the user's incoming mail file. This is done with the `read-incoming-messages` function, listed in Figure 5.9.

The actual web page comes from the `classify-form` function, listed in Figure 5.10. This form is created with the `html` macro, which allows us to write a "lispified" html template

```
(defun classify-and-write-files (user spam-indices nonspam-indices)
  (let (remaining-messages)
    (mp:with-process-lock ((user-process-lock user))
      (let (*read-eval*)
        (let ((messages (read-incoming-messages user)))
          (when messages
            (let ((incoming-buffer-pathname (sys:make-temp-file-name))(count -1)
                  (spams-pathname
                   (progn (ensure-directories-exist (string-append "/var/mail/" user ".data/corpi/"))
                          (string-append "/var/mail/" user ".data/corpi/spams")))
                  (nonspams-pathname (string-append "/var/mail/" user ".data/corpi/nonspams")))
              (with-open-file (spams spams-pathname
                                     :direction :output :if-exists :append :if-does-not-exist :create)
                (with-open-file (nonspams nonspams-pathname
                                          :direction :output :if-exists :append :if-does-not-exist :create)
                  (with-open-file (popqueue (string-append "/var/mail/" user)
                                            :direction :output :if-exists :append :if-does-not-exist :create)
                    (with-open-file (incoming-buffer incoming-buffer-pathname
                                                     :direction :output :if-exists :supersede :if-does-not-exist :create)
                      (dolist (message messages)
                        (let ((index (getf message :index)))
                          (cond ((gethash index spam-indices)
                                 (record-tokens (mapcar #'(lambda(token)(intern token *token-package*))
                                                        (getf message :tokens)) :spam user)
                                 (print message spams))
                                ((gethash index nonspam-indices)
                                 (record-tokens (mapcar #'(lambda(token) (intern token *token-package*))
                                                        (getf message :tokens)) :nonspam user)
                                 (print message nonspams)
                                 (pop-format message popqueue))
                                (t
                                 (setf (getf message :index) (incf count))
                                 (push message remaining-messages)
                                 (print message incoming-buffer)))))))))
              (sys:copy-file incoming-buffer-pathname (string-append "/var/mail/incoming/" user)
                             :overwrite t)
              (delete-file incoming-buffer-pathname))))))
    (nreverse remaining-messages)))
```

Figure 5.8: Function to distribute messages to the appropriate files

with embedded Lisp code which will be evaluated. In this manner, we create the html page including a form whose action (response URL) invokes the same `classify` function as described above. We then group the messages, based on their pre-computed score, into a list of expected spams and expected nonspams. Finally we present the headers from these messages to the user, each with its radio button pre-selected appropriately if we have a sufficient confidence level of the spam/nonspam probability of the message.

When the user presses the "Submit" button, the form is submitted and the same `classify` function will respond, distributing messages into the appropriate files. This process is repeated indefinitely as long as the user continues pressing the "Submit" button.

```
(defun read-incoming-messages (user)
  (fetch-mail user)
  (let (result (*read-eval* nil) (count -1)
        (incoming-pathname (string-append "/var/mail/incoming/" user)))
    (when (probe-file incoming-pathname)
      (with-open-file (in incoming-pathname)
        (do ((message (read in nil nil) (read in nil nil)))
            ((null message) (nreverse result))
          (setf (getf message :index) (incf count))
          (push message result))))))
```

Figure 5.9: Reading incoming messages

```
(defun classify-form (messages user)
  (html
   (:html
    (:head (:title "Classify Incoming Mail for " (:princ user)))
    (:body (:h2 (:center "Classify Incoming Mail for " (:princ user)))
           ((:form :action "/classify" :method :post)
            (:p ((:input :type :submit :value "Classify!")))
            (:center
             ((:input :type :hidden :name :user :value user))
             ((:table :bgcolor :black :cellpadding 1 :cellspacing 1)
              (let ((grouped-messages (list nil nil)))
                (mapc #'(lambda(message)
                          (if (> (getf message :score) 0.8)
                              (push message (second grouped-messages))
                            (push message (first grouped-messages)))) messages)
                (setq grouped-messages (append (nreverse (first grouped-messages))
                                               (nreverse (second grouped-messages))))
                (when grouped-messages
                  (html ((:tr :bgcolor :yellow) (:th "Bad") (:th "Good") (:th "From") (:th "Subject") (:th "Date") (:th "Score"))))
                (dolist (message grouped-messages)
                  (let ((headers (getf message :headers))
                        (score (getf message :score)))
                    (let ((from (rest (assoc "from" headers :test #'string-equal)))
                          (subject (rest (assoc "subject" headers :test #'string-equal)))
                          (date (rest (assoc "date" headers :test #'string-equal))))
                      (html ((:tr :bgcolor (if (> score 0.8) :pink "#aaffaa"))
                             ((:td :bgcolor :red)
                              ((:input :type :radio :name (format nil "spamp-~a" (getf message :index))
                                       :value :spam :if* (> score 0.8) :checked :checked)))
                             ((:td :bgcolor :green)
                              ((:input :type :radio :name (format nil "spamp-~a" (getf message :index))
                                       :value :non-spam :if* (< score 0.5) :checked :checked)))

                             (:td (:small (:princ (string-append (subseq subject 0 (min (length subject) *max-subject-length*))
                                                                 (if (> (length subject) *max-subject-length*) "..." ""))))
                             (:td (:small (:princ (short-date date))))
                             (:td (:princ score)))))))))))
           (:p ((:input :type :submit :value "Classify!")))))))
```

Figure 5.10: Generating the actual web page

Appendix A

Squeakymail with Genworks' GDL/GWL

Genworks International produces and sells a Knowledge Base system based on Allegro CL called General-purpose Declarative Language. Bundled with GDL is also GWL, Generative Web Language, an integrated web application server which runs in conjunction with GDL and AllegroServe.

GDL and GWL can also generate and manipulate 3D geometric entities, giving you additional power in a full-featured toolset for creating engineering and business applications.

For an introduction to GDL/GWL, please see the GDL Tutorial, available in PDF format from:

`http://www.genworks.com/papers/gdl-tutorial.pdf`

In this chapter we present an alternative classifying/browsing interface for Squeakymail, based on GDL/GWL. This version of the interface contains some significant features not found in the version found in Chapter 5:

- Secure user login, provided by a separate GWL module
- Ability to preview the message bodies in a linked page
- Convenient interfaces for a user to add, update, and delete remote POP accounts and explicit message filters.
- Facility for "plugging in" a user's Squeakymail access into a larger web-based application framework, consisting of for example, calendaring and personal task management and planning apps.

A.1 Toplevel Squeakymail Home Page

The code in Figure A.1 shows the object definition corresponding to the Squeakymail application home page. Note the `customer-row` being passed in as an input-slot. This customer-row is a database record, provided by a separate GWL module, containing information on an authenticated user.

The child objects `classify`, `pop-accounts`, and `filters` are to be presented as hyperlinks to specific pages within the main Squeakymail page. `Classify` allows is a page allowing the user to browse and classify mail. `Pop-accounts` allows the user to manage remote POP accounts. Finally, `filters` allows the user to manage explicit filters, which allow messages to bypass the standard Baysian filtering mechanism.

The view defined in Figure A.2 defines a `main-sheet` output function, which generates the actual HTML for the page.

A.2 Page for Browsing and Classifying

Figure A.3 builds up a sequence of current messages, each potentially to be displayed on its own sub-page, and the view defined in A.4 defines a `main-sheet` output function which processes incoming mail, then calls the `classify-form` output function to emit the actual HTML for the classification form.

The `message` object defined in Figure A.5 is a simple message taking inputs of `from`, `subject`, `to`, `body`, `score`, and `scored-tokens`. Each message can potentially be rendered in HTML using the `main-sheet` output function defined in the view in Figure A.6.

```
(define-object assembly (base-html-sheet)
  :input-slots
  (customer-row)

  :computed-slots
  ((local-login (the customer-row login)))

  :objects
  ((classify :type 'classify
             :strings-for-display "Check and Classify"
             :pass-down (local-login))

   (pop-accounts :type 'html-sql-table
                 :sql-server *local-connection*
                 :table-name 'pop-account
                 :fixed-fields (list :local-login (the customer-row login))
                 :strings-for-display "Pop Accounts"
                 :columns-and-names
                 `(("host" "Mail Server")
                   ("login" "Username")
                   ("password" "Password")))

   (filters :type 'html-sql-table
            :sql-server *local-connection*
            :table-name 'filters
            :strings-for-display "Filters"
            :fixed-fields (list :local-login (the customer-row login))
            :columns-and-names
            `(("source" "Source"
               :choices (:explicit ("Subject" "To" "From" "Body")))
              ("expression" "Expression" :size 45)
              ("destination" "Put In"
               :choices (:explicit ("Spam" "Nonspam")))))))
```

Figure A.1: Object for Squeakymail Home Page

```
(define-view (html-format assembly)()

  :output-functions
  ((main-sheet
    ()
    (html (:html (:head
                  (:title "Squeakymail - Enjoy a squeaky-clean email experience"))
                 (:body (when *developing?* (the write-development-links))
                        (when (the return-object) (the write-back-link))
                        (:center (:h2 "Welcome to Squeakymail")
                                 (:i "Enjoy a Squeaky-Clean Email Experience"))

                        (:p (the classify (write-self-link)))
                        (:p (the filters (write-self-link)))
                        (:p (the pop-accounts (write-self-link)))

                        (the write-standard-footer)))))))
```

Figure A.2: Output Function for Squeakymail Home Page

```
(define-object classify (base-html-sheet)

  :input-slots
  (local-login)

  :computed-slots
  ((message-list   (progn (fetch-mail (the local-login))
                          (read-incoming-messages (the local-login)))))

  :objects
  ((messages :type 'message
             :sequence (:size (length (the message-list)))
             :data (nth (the-child index) (the message-list))
             :headers (getf (the-child :data) :headers)
             :scored-tokens (getf (the-child :data) :scored-tokens)
             :from (or (rest (assoc "from" (the-child headers)
                                    :test #'string-equal)) "")
             :subject (or (rest (assoc "subject" (the-child headers)
                                       :test #'string-equal)) "")
             :to (or (rest (assoc "to" (the-child headers)
                                  :test #'string-equal)) "")
             :date (or (rest (assoc "date" (the-child headers)
                                    :test #'string-equal)) "")
             :body (getf (the-child :data) :body)
             :score (getf (the-child :data) :score))))
```

Figure A.3: Object for Squeakymail Classification Page

```
(define-view (html-format classify)()

  :output-functions
  ((main-sheet
    ()
    (let ((query-plist (the query-plist)))
      (classify-and-write-files query-plist (the local-login)))
    (the (restore-attribute-defaults! (list :message-list :query-plist)))

    (write-the (classify-form (list-elements (the messages)) (the local-login))))

   (classify-form
    (messages user)
    (html
     (:html
      (:head (:title (if messages (html "Classify Incoming Mail for ")
                         (html "No Mail for ")) (:princ user)))
      (:body (:p (when gwl:*developing?* (the write-development-links)))
             (:h2 (:center (if messages (html "Classify Incoming Mail for ")
                               (html "No Mail for ")) (:princ user)))
             ((:form :action "/answer" :method :post)
              (:p (the write-back-link))
              (:p ((:input :type :submit :value (if messages "Classify!" "Check Again!"))))
              (:center
               (the (:write-form-fields))
               ((:table :bgcolor :black :cellpadding 1 :cellspacing 1)
                (let ((grouped-messages (list nil nil nil)))
                  (mapc #'(lambda(message)
                            (let ((score (the-object message score)))
                              (cond ((null score) (push message (first grouped-messages)))
                                    ((> score 0.8) (push message (third grouped-messages)))
                                    (t (push message (second grouped-messages))))))) messages)
                  (setq grouped-messages (append (nreverse (first grouped-messages))
                                                 (nreverse (second grouped-messages))
                                                 (nreverse (third grouped-messages))))
                  (when grouped-messages
                    (html ((:tr :bgcolor :yellow) (:th "Bad") (:th "Good") (:th "From")
                                                  (:th "Subject") (:th "Date") (:th "Score"))))
                  (dolist (message grouped-messages)
                    (let ((from (the-object message from))
                          (subject (the-object message subject))
                          (date (the-object message date))
                          (score (the-object message score)))
                      (html ((:tr :bgcolor (cond ((null score) (gethash :sky-summer *color-table*))
                                                 ((> (the-object message score) 0.8) :pink) (t "#aaffaa")))
                             ((:td :bgcolor :red)
                              ((:input :type :radio :name (format nil "spamp-~a" (the-object message index))
                                       :value :spam :if* (and score (> (the-object message score) 0.8)) :checked :checked)))
                             ((:td :bgcolor :green)
                              ((:input :type :radio :name (format nil "spamp-~a" (the-object message index))
                                       :value :non-spam :if* (and score (< (the-object message score) 0.5)) :checked :checked)))
                             (:td (:small (:princ-safe (excl:replace-regexp from """ "")))))
                             (:td (:small (the-object message
                                                      (write-self-link
                                                       :display-string
                                                       (with-output-to-string(*html-stream*)
                                                         (html
                                                          (:princ-safe
                                                           (string-append (subseq subject 0 (min (length subject)
                                                                                                 *max-subject-length*))
                                                                          (if (> (length subject) *max-subject-length*)
                                                                              "..." "")))))))))
                             (:td (:small (:princ-safe (short-date date))))
                             (:td (:princ score)))))))))
             (when messages (html (:p ((:input :type :submit :value "Classify!"))))))))))
```

Figure A.4: Output Function for Classification Page

```
(define-object message (base-html-sheet)
  :input-slots
  (from subject to body score scored-tokens))
```

Figure A.5: Object for a Single Message

```
(define-view (html-format message)()
  :output-functions
  ((main-sheet
    ()
    (html
     (:html
      (:head
       (:title (:princ (the subject)))
       (:body
        (:p (the write-back-link))
        (:p (:table (:tr (:td "From") (:td (:princ-safe (the from))))
                    (:tr (:td "To") (:td (:princ-safe (the to))))
                    (:tr (:td "Subject") (:td (:princ-safe (the subject))))
                    (:tr (:td "Score") (:td (:princ-safe (the score))))
                    (:tr ((:td :colspan 2) (:pre (:princ (the body)))))))
        (when *show-tokens?*
          (html (:p ((:table :bgcolor :black)
                     (dolist (pair (subseq (the scored-tokens) 0
                                           (min (length (the scored-tokens)) 50)))
                       (html ((:tr :bgcolor (if (> (second pair) 0.5)
                                                :pink "#aaffaa"))
                              (:td (:princ-safe (first pair)))
                              (:td (format *html-stream* "~2,7f"
                                           (second pair))))))))))))))))
```

Figure A.6: Output Function for Displaying a Message

Appendix B

Bibliography

This Bibliography should also be considered as a Recommended Reading list.

- ANSI Common Lisp, by Paul Graham. Prentice-Hall, Inc, Englewood Cliffs, New Jersey. 1996.
- Common Lisp, the Language, 2nd Edition, by Steele, Guy L., Jr., with Scott E. Fahlman, Richard P. Gabriel, David A. Moon, Daniel L. Weinreb, Daniel G. Bobrow, Linda G. DeMichiel, Sonya E. Keene, Gregor Kiczales, Crispin Perdue, Kent M. Pitman, Richard C. Waters, and Jon L. White. Digital Press, Bedford, MA. 1990.
- ANSI CL Hyperspec, by Kent M. Pitman. Available at

  `http://www.xanalys.com`

- Learning Gnu Emacs, by Eric Raymond. O'Reilly and Associates. 1996.
- Association of Lisp Users website, many other resources and references, at `http://www.alu.org`
- Usenet Newsgroup `comp.lang.lisp`
- Successful Lisp, by David Lamkin, available at

  `http://psg.com/~dlamkins/left/sl/sl.html`

- Franz Website (Commercial CL Vendor), at

  `http://www.franz.com`

- Xanalys Lispworks Website (Commercial CL Vendor), at

  `http://www.lispworks.com`

- Symbolics Website (Commercial CL Vendor), at

`http://www.symbolics.com`

- Digitool Website (Macintosh Common Lisp Vendor), at

  `http://www.digitool.com`

- Corman Lisp Website (Commercial CL Vendor), at

  `http://www.corman.net`

- Genworks International Website (CL-based tools, services, and pointers to other resources) at

  `http://www.genworks.com`

- Knowledge Technologies International Website, at

  `http://www.ktiworld.com`

- Tulane Lisp Tutorial, available at

  `http://www.cs.tulane.edu/www/Villamil/lisp/lisp1.html`

- Texas A&M Lisp Tutorial, available at

  `http://grimpeur.tamu.edu/colin/lp/`

Appendix C

Emacs Customization

C.1 Lisp Mode

When you edit a Lisp file in Emacs, Emacs should automatically put you into Lisp Mode or Common Lisp Mode. You will see this in the status bar near the bottom of the screen. Lisp Mode gives you many convenient commands for working with List expressions, otherwise known as Symbolic Expressions, or *s-expressions*, or *sexp*'s for short.

For a complete overview of Lisp Mode, issue the command `M-x describe-mode` in Emacs. Table C.1 describes a sampling of the commands available in Lisp Mode[1].

C.2 Making Your Own Keychords

You may wish to automate some commands which do not already have keychords associated with them. The usual way to do this is to invoke some `global-set-key` commands in the `.emacs` file (Emacs' initialization file) in your home directory. Here are some example entries you could put in this file, along with comments describing what they do:

[1]Some of these commands are actually available in Fundamental mode as well

Keychord	*Action*	*Emacs Function*
`C-M-space`	Mark sexp starting at point	`mark-sexp`
`M-w`	Copy marked expression	`kill-ring-save`
`C-y`	Paste copied expression	`yank`
`C-M-k`	Kill the sexp starting at point	`kill-sexp`
`C-M-f`	Move point forward by one sexp	`forward-sexp`
`C-M-b`	Move point backward by one sexp	`backward-sexp`
`M-/`	Dynamically expand current word (cycles through choices on repeat)	`dabbrev-expand`

Table C.1: Common Commands of the Emacs-Lisp Interface

```
;; Make C-x & switch to the *common-lisp* buffer
(global-set-key "\C-x&" '(lambda() (interactive)
                           (switch-to-buffer "*common-lisp*")))

;; Make function key 5 (F5) start or visit an OS shell.
(global-set-key [f5] '(lambda() (interactive) (shell)))

;; Make sure M-p functions to scroll through previous commands.
(global-set-key "\M-p" 'fi:pop-input)

;; Make sure M-n functions to scroll through next commands.
(global-set-key "\M-n" 'fi:push-input)

;; Enable the hilighting of selected text to show up.
(transient-mark-mode 1)
```

C.3 Keyboard Mapping

If you plan to spend significant time working with your keyboard, consider investing some time in setting it up so you can use it effectively. As a minimum, you should make sure your Control and Meta keys are set up so that you can get at them without moving your hands away from the "home keys" on the keyboard. This means that:

- The Control key should be to the left of the "A" key, assessible with the little finger of your left hand.
- The Meta keys should be on each side of the space bar, accessible by tucking your right or left thumb under the palm of your hand.

If you are working on an XWindow terminal (e.g. a Unix or Linux machine, or a PC running an X Server such as Hummingbird eXceed) you can customize your keyboard with the `xmodmap` Unix command. To use this command, prepare a file called `.Xmodmap` with your desired customizations, then invoke this file with the command

```
xmodmap .Xmodmap
```

Here is an example `.Xmodmap` file for a PC keyboard, which makes the Caps Lock key function as Control, and ensures that both Alt keys will function as Meta:

```
remove Lock = Caps_Lock
keysym Caps_Lock = Control_L
add Control = Control_L
keysym Alt_L = Meta_L Alt_L
```

For PCs running Microsoft Windows, the Windows Control Panel should contain options for customizing the keyboard in a similar manner.

Appendix D

Afterword

D.1 About This Book

This book was typeset using a Lisp-based markup language. For the printed output, the Lisp-based markup language generated LaTeX code and was typeset using LaTeX by Leslie Lamport, which is built atop TeX by Donald Knuth.

D.2 Acknowledgements

I have received inspiration and instruction from many enabling me to write this book. I would like to thank, in no particular order, John McCarthy, Erik Naggum of Naggum Software and comp.lang.lisp, John Foderaro of Franz Inc., Professor John Laird who taught me CL in college; John Mallery and Paul Graham, who taught me that CL is *the* platform to be using for server applications, James Russell, Andrew Wolven, I would especially like to thank Lisa Fettner, Dr. Sheng-Chuan Wu, and others at Franz Inc for their eagle-eyed proofreading, Peter Karp from Stanford Research Institute for creating the original outline, and Hannu Koivisto for helping to improve the PDF output. Also thank you to Michael Drumheller of the Boeing Co. for providing feedback on the first edition. "Thank you" to my wife Kai, for putting up with my antics while I was writing this (and in general). I would also like to thank Fritz Kunze of Franz Inc. for telling me to "go write a book."

Any remaining errors in this version are of course my own.

D.3 About the Author

David J. Cooper has been Chief Engineer with Genworks International since November 1997. His principal activity is working with large manufacturing companies, helping them to solve engineering and manufacturing problems, mostly with CL-based tools. He has also led instruction courses at General Motors, Visteon Automotive, Raytheon Aircraft, Ford Motor Company, Knowledge Technologies International, and others. Prior to his career with Genworks, Mr. Cooper worked with Ford Motor Company, mostly on deployment of CL-based systems in the field of mechanical engineering and manufacturing. He also has software development experience in the CAD and database industries. He received his M.S.

degree in Computer Science from the University of Michigan in 1993, and holds a B.S. in Computer Science and a B.A. in German from Michigan as well. His current interests include marketing and promoting Genworks' GDL/GWL Knowledge Base product offering. He is available for consultation and product inquiries through Genworks International at http://www.genworks.com, david.cooper@genworks.com, +1 (248) 910-0912.

Index