Loving Lisp, or the Savvy Programmer's Secret Weapon

Mark Watson

Copyright 2002, Mark Watson, all rights reserved.

Version 0.7   Last updated: September 18, 2002

This document can be redistributed in its original and un-altered form.

Please check Mark Watson's web site www.markwatson.com for updated versions.

Requests from the author

This web book may be distributed freely in an unmodified form. Please report any errors to markw@markwatson.com.

I live in a remote area, the mountains of Northern Arizona and work remotely via the Internet. Although I really enjoy writing Open Content documents like this Web Book and working on Open Source projects, I earn my living as a Java and Common Lisp consultant. Please keep me in mind for consulting jobs! Also, please read my resume and consulting terms at www.markwatson.com.

If you enjoy this free Web Book and can afford to, please consider making a small donation of $2.00 using the PayPal link on www.markwatson.com or sending a check or cash to:

Mark Watson
120 Farmer Brothers Drive
Sedona, AZ 86336

Copyright 2002, Mark Watson. All rights reserved.

ACKNOWLEDGEMENTS.. 4

1. INTRODUCTION.. 5

1.1 Why did I write this book? 5

1.2 Free software tools for Common Lisp programming.. 5

1.3 How is Lisp different from languages like Java and C++? 6

1.4 Advantages of working in a Lisp environment 7

1.5 Getting Started with CLISP 8

2. THE BASICS OF LISP PROGRAMMING.. 11

2.1 Symbols.. 15

2.2 Operations on Lists 15

2.3 Using arrays and vectors.. 19

2.4 Using Strings.. 20

2.5 Using hash tables.. 23

2.6 Using Eval to evaluate Lisp Forms 26

2.7 Using a text editor to edit Lisp source files.. 26

2.8 Recovering from Errors 27

2.9 Garbage collection 28

2.10 Loading your Working Environment Quickly.. 28

3. DEFINING LISP FUNCTIONS 30

3.1 Using lambda forms.. 31

3.2 Using recursion.. 32

3.3 Closures 33

4. DEFINING COMMON LISP MACROS.. 35

4.1 Example macro.. 35

4.2 Using the splicing operator 36

Copyright 2002, Mark Watson. All rights reserved.

4.3 Using macroexpand-1 36

5. USING COMMON LISP LOOP MACROS 37

5.1 dolist 37

5.2 dotimes 37

5.3 do 38

5.4 loop 39

6. INPUT AND OUTPUT 40

6.1 The Lisp read and read-line functions 40

6.2 Lisp printing functions 42

7. COMMON LISP PACKAGE SYSTEM 45

8. COMMON LISP OBJECT SYSTEM - CLOS 47

8.1 Example of using a CLOS class 47

8.2 Implementation of the HTMLstream class 48

8.3 Other useful CLOS features 51

9. NETWORK PROGRAMMING 52

9.1 An introduction to sockets 52

9.2 A server example 53

9.3 A client example 54

9.4 An Email Client 55

INDEX 59

Copyright 2002, Mark Watson. All rights reserved.

Acknowledgements

I would like to thank the following people who made suggestions for improving this web book:

Sam Steingold, Andrew Philpot, Kenny Tilton, Mathew Villeneuve, Eli Draluk, Erik Winkels, Adam Shimali, and Paolo Amoroso.

I would like to thank Paul Graham for coining the phrase "The Secret Weapon" (in his excellent paper "Beating the Averages") in discussing the advantages of Lisp.

Copyright 2002, Mark Watson. All rights reserved.

1. Introduction

This book is intended to get you, the reader, programming quickly in Common Lisp. Although the Lisp programming language is often associated with artificial intelligence, this is not a book on artificial intelligence.

This free web book is distributed as a ZIP file that also contains a directory **src** containing Lisp example programs and an HTML file **readme.html** that contains web links for Common Lisp resources on the Internet.

1.1 Why did I write this book?

Why the title “Loving Lisp”? Simple! I have been using Lisp for over 20 years and seldom do I find a better match between a programming language and the programming job at hand. I am not a total fanatic on Lisp however. I like Java for server side programming, and the few years that I spent working on Nintendo video games and virtual reality systems for SAIC and Disney, I found C++ to be a good bet because of stringent runtime performance requirements. For some jobs, I find the logic-programming paradigm useful: I also enjoy the Prolog language.

In any case, I love programming in Lisp, especially the industry standard Common Lisp.

As programmers, we all (hopefully) enjoy applying our experience and brains to tackling interesting problems. My wife and I recently watched a two-night 7-hour PBS special “Joseph Campbell, and the Power of Myths”. Campbell, a college professor for almost 40 years, said that he always advised his students to “follow their bliss” and to not settle jobs and avocations that are not what they truly want to do. That said I always feel that when a job calls for using Java, C++, or perhaps Prolog, that even though I may get a lot of pleasure from completing the job, I am not following my bliss.

My goal in this book is to introduce you to my favorite programming language, Common Lisp. I assume that you already know how to program in another language, but if you are a complete beginner, you can still master the material in this book with some effort. I challenge you to make this effort.

1.2 Free software tools for Common Lisp programming

There are several Common Lisp compilers and runtime tools available for free on the web:

> CLISP - licensed under the GNU GPL and is available for Windows, Macintosh, and Linux/Unix

Copyright 2002, Mark Watson. All rights reserved.

OpenMCL - licensed under the GNU LGPL license and is available for Mac OS X
CMU Common Lisp - a public domain style license and is available for several types of Unix and Linux (Pentium, SPARK, and ALPHA – with some work towards PowerPC support)
Steel Bank Common Lisp - derived from CMU Common Lisp

There are also fine commercial Common Lisp products:

Xanalys LispWorks - high quality and reasonably priced system for Windows and Linux. No charge for distributing compiled applications.
Franz Allegro Lisp - high quality and high cost.
MCL - Macintosh Common Lisp. I used this Lisp environment in the late 1980s, and it seems to get better every year.

For working through this book, I will assume that you are using CLISP.

1.3 How is Lisp different from languages like Java and C++?

This is a trick question! Lisp is more similar to Java than C++ so we will start by comparing Lisp and Java.

In Java, variables are strongly typed while in Common Lisp values are strongly typed. For example, consider the Java code:

```
Float x = new Float(3.14f);
String s = " the cat ran " ;
Object any_object = null;
any_object = s;
x = s;  // illegal: generates a compilation error
```

Here, in Java, variables are strongly typed so a variable x of type Float can not legally be assigned a string value: this would generate a compilation error.

Java and Lisp share the capability of automatic memory management. In either language, you can create new data structures and not worry about freeing memory when the data is no longer used, or to be more precise, no longer referenced.

Common Lisp is an ANSI standard language. Portability between different Common Lisp implementations and on different platforms is very good. I do most of my Lisp development on a Mac running OS X using CLISP, Emacs, and ILISP; when I need to deliver executables for delivery under Windows or Linux, I simply re-compile the code using Xanalys LispWorks on those alternative platforms.

Copyright 2002, Mark Watson. All rights reserved.

ANSI Common Lisp was the first object oriented language to become an ANSI standard language. The Common Lisp Object System (CLOS) is probably the best platform for object oriented programming.

The CLOCC project provides portable Common Lisp utilities for a wide variety of Common Lisp implementations. Check out the web site http://clocc.sourceforge.net.

In C++ programs, a common bug that affects a program's efficiency is forgetting to free memory that is no longer used (in a virutal memory system, the effect of a program's increasing memory usage is usually just poorer system performance but can lead to system crashes or failures if all available virtual memory is exhausted.) A worse type of C++ error is to free memory and then try to use it. Can you say "program crash"? C programs suffer from the same types of memory related errors.

Since computer processing power is usually much less expensive than the costs of software development, it is almost always worth while to give up a few percent of runtime efficiency and let the programming environment of runtime libraries manage memory for you. Languages like Lisp, Python, and Java are said to perform automatic garbage collection.

I have written six books on Java, and I have been quoted as saying that for me, programming in Java is about twice as efficient (in terms of my time) than programming in C++. I base this statement on approximately ten years of C++ experience on projects for SAIC, PacBell, Angel Studios, Nintendo, and Disney. I find Common Lisp to be about twice as efficient (again, in terms of my time) than Java.

What do I mean by programmer efficiency? Simple: for a given job, how long does it take me to design, code, debug, and later maintain the software for a given task.

1.4 Advantages of working in a Lisp environment

We will soon see in this book that Lisp is not just a language; it is also a programming environment and runtime environment.

The beginning of this book introduces the basics of Lisp programming. In later chapters, we will develop interesting and non-trivial programs in Common Lisp that I argue would be more difficult to implement in other languages and programming environments.

The big win in programming in a Lisp environment is that you can set up an environment and interactively write new code and test new code in small pieces. We will cover programming with large amounts of data in a separate chapter later, but let me share a use case for work that I do every day that is far more efficient in Lisp:

Most of my Lisp programming is to write commercial natural language processing (NLP) programs for my company www.knowledgebooks.com. My Lisp NLP code uses a huge

Copyright 2002, Mark Watson. All rights reserved.

amount of memory resident data; for example: hash tables for different types of words, hash tables for text categorization, 200,000 proper nouns for place names (cities, counties, rivers, etc.), and about 40,000 common first and last names of various nationalities. If I was writing our NLP products in C++, I would probably use a relational database to store this data because if I read all of this data into memory for each test run of a C++ program, I would wait 30 seconds every time that I ran a program test. I do most of my programming on a Macintosh using OS X and my programming environment is free and powerful: Emacs with CLISP. When I start working, I do have to load the linguistic data into memory one time, but then can code/test/code/test... for hours with no startup overhead for reloading the data that my programs need to run. Because of the interactive nature of Lisp development, I can test small bits of code when tracking down problem in the code.

It is a personal preference, but I find the combination of the stable Common Lisp language and an iterative Lisp programming environment to be much more productive than, for example, the best Java IDEs (e.g., IntelliJ Idea is my favorite) and Microsoft's VisualStudio.Net (which admittedly makes writing web services almost trivial).

1.5 Getting Started with CLISP

As we discussed in the introduction, there are many different Lisp programming environments that you can choose from. I recommend a free set of tools: Emacs, ILISP, and CLISP. Emacs is a fine text editor that is extensible to work well with many programming languages and document types (e.g., HTML and XML). ILISP is a Emacs extension package that greatly facilitates Lisp development. CLISP is a robust Common Lisp compiler and runtime system. I will not discuss the use of Emacs and ILISP in this book. If you either already use Emacs or do not mind spending the effort to learn Emacs, then search the web first for an Emacs tutorial ("Emacs tutorial") and then for information on ILISP (web search for: "Emacs ILISP CLISP" - assuming that you will start with the CLISP Common Lisp compiler).

If you do not already have CLISP installed on your computer, visit the web site **http://CLISP.sourceforge.net/** and download CLISP for your computer type.

Here, we will assume that under Windows, Unix, Linux, or Mac OS X that you will use one command window to run CLISP and a separate editor that can edit plain text files.

When we start CLISP, we see a introductory message crediting the people who work on CLISP and then an input prompt. We will start with a short tutorial, walking you through a session using CLISP (other Common LISP systems are very similar). Assuming that CLISP is installed on your system, start CLISP by running the CLISP program:

```
[localhost:~] markw% CLISP
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
```

Copyright 2002, Mark Watson. All rights reserved.

```
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8          8  8
        |            8     o   8           8     o    8  8
  ------+------       ooooo    8oooooo  ooo8ooo   ooooo   8

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2001

;; Loading file /Users/markw/.CLISPrc ...
;; Loading of file /Users/markw/.CLISPrc is finished.
[1]>
```

I customize my CLISP environment by placing my own initialization code in a special file **.clisprc** that can be placed in your home directory. We will discuss customization later. The last line **[1]>** indicates that CLISP is ready to process command number 1. We will start by defining a variable and performing a few simple operations to introduce you to working in a Lisp listener interface. A Lisp listener interface is a loop that waits for you to type in an expression, evaluates and prints the value of that expression, then waits for you to type another command. We will continue with our introduction to using a Lisp listener:

```
[1]> (defvar x 1.0)
X
[2]> x
1.0
[3]> (+ x 1)
2.0
[4]> x
1.0
[5]> (setq x (+ x 1))
2.0
[6]> x
2.0
[7]> (setq x "the dog chased the cat")
"the dog chased the cat"
[8]> x
"the dog chased the cat"
[9]>
```

We started by defining a new variable x. Notice how the value of the **defvar** macro is the symbol that is defined. The Lisp reader prints X capitalized because symbols are made upper case (we will look at the exception later).

In Lisp, a variable can reference any data type. We start by assigning a floating point value to the variable x, using the + function to add 1 to x, using the **setq** function to change the value of x first to another floating point value and finally setting x to a string value. One thing that you will have noticed: function names always occur first, then the arguments to a function. Also, parenthesis is used to separate expressions.

Copyright 2002, Mark Watson. All rights reserved.

I learned to program Lisp in 1974 and my professor half-jokingly told us that Lisp was an acronym for "Lots-of Irritating Superfluous Parenthesis". There may be some truth in this when you are just starting with Lisp programming, but you will quickly get used to the parenthesis, especially if you use an editor like Emacs that automatically indents Lisp code for you and highlights the opening parenthesis for every closing parenthesis that you type. Many other editors also support coding in Lisp but we will only cover the use of Emacs in this web book. Newer versions of Emacs and XEmacs also provide colored syntax highlighting of Lisp code. While I use XEmacs and CLISP for my work (because only XEmacs provides color syntax highlighting on my system), I will provide screen shots of Emacs and CLISP in this web book. Take your pick and choose the text editor that works best for you; I prefer Emacs (or XEmacs).

Before you proceed to the next chapter, please take the time to install CLISP on your computer and try typing some expressions into the Lisp listener. If you get errors, or want to quit, try using the quit function:

```
[10]> (+ 1 2 3 4)
10
[11]> (quit)
Bye.
```

Copyright 2002, Mark Watson. All rights reserved.

2. The Basics of Lisp Programming

Although we will use CLISP in the web book, any Common Lisp environment will do fine. In the Introduction, we saw the top-level Lisp prompt and how we could type any expression that would be evaluated:

```
[1]> 1
1
[2]> 3.14159
3.14159
[3]> "the dog bit the cat"
"the dog bit the cat"
[4]> (defun my-add-one (x)
(+ x 1))
MY-ADD-ONE
[5]> (my-add-one -10)
-9
```

CLISP keeps a counter of how many expressions have been evaluated; this number appears in square brackets before the > prompt. For the rest of this web book, we will omit the expression counter.

Notice that when we defined the function **my-add-one**, we split the definition over two lines. The top level Lisp evaluator counts parentheses and considers a form to be complete when the number of closing parentheses equals the number of opening parentheses. When we evaluate a number (or a variable), there are no parentheses, so evaluation proceeds when we hit a new line (or carriage return).

The Lisp reader by default tries to evaluate any form that you enter. There is a reader macro ' that prevents the evaluation of an expression. You can either use the ' character or "quote":

```
> (+ 1 2)
3
> '(+ 1 2)
(+ 1 2)
> (quote (+ 1 2))
(+ 1 2)
>
```

Lisp supports both global and local variables. Global variables can be declared using **defvar**:

```
> (defvar *x* "cat")
*X*
> *x*
"cat"
> (setq *x* "dog")
"dog"
> *x*
```

Copyright 2002, Mark Watson. All rights reserved.

```
"dog"
> (setq *x* 3.14159)
3.14159
> *x*
3.14159
```

One thing to be careful of when defining global variables with **defvar**: the declared global variable is dynamically scoped. We will discuss dynamic versus lexical coping later, but for now a warning: if you define a global variable avoid redefining the same variable name inside functions. Lisp programmers usually use a global variable naming convention of beginning and ending dynamically scoped global variables with the * character. If you follow this naming convention and also do not use the * character in local variable names, you will stay out of trouble. For convenience, I do not always follow this convention in short examples in this book.

Lisp variables have no type. Rather, values assigned to variables have a type. In this last example, the variable x was set to a string, then to a floating-point number. Lisp types support inheritance and can be thought of as a hierarchical tree with the type t at the top. (Actually, the type hierarchy is a DAG, but we can ignore that for now.) Common Lisp also has powerful object oriented programming facilities in the Common Lisp Object System (CLOS) that we will discuss in a later chapter.

Here is a partial list of types (note that indentation denotes being a subtype of the preceding type):

```
t  [top level type (all other types are a sub-type)]
     sequence
          list
          array
               vector
                    string
     number
          float
          rational
               integer
               ratio
          complex
     character
     symbol
     structure
     function
     hash-table
```

We can use the **typep** function to test the type of value of any variable or expression:

```
> (setq x '(1 2 3))
(1 2 3)
> (typep x 'list)
T
> (typep x 'sequence)
T
> (typep x 'number)
NIL
```

Copyright 2002, Mark Watson. All rights reserved.

```
> (typep (+ 1 2 3) 'number)
T
>
```

A useful feature of the CLISP (and all ANSI standard Common Lisp implementations) top-level listener is that it sets * to the value of the last expression evaluated. For example:

```
> (+ 1 2 3 4 5)
15
> *
15
> (setq x *)
15
> x
15
```

All Common Lisp environments set * to the value of the last expression evaluated.

Frequently, when you are interactively testing new code, you will call a function that you just wrote with test arguments; it is useful to save intermediate results for later testing. It is the ability to create complex data structures and then experiment with code that uses or changes these data structures that makes Lisp programming environments so effective.

Common Lisp is a lexically scoped lexically scoped language that means that variable declarations and function definitions can be nested and that the same variable names can be used in nested let forms; when a variable is used, the current let form is searched for a definition of that variable and if it is not found, then the next outer let form is searched. Of course, this search for the correct declaration of a variable is done at compile time so there need not be extra runtime overhead. Consider the following example in the file **nested.lisp** (all example files are in the **src** directory that is distributed with the PDF file for this web book):

```
(let ((x 1)
      (y 2))
  ;; define a test function nested inside a let statement:
  (defun test (a b)
    (let ((z (+ a b)))
      ;; define a helper function nested inside a let/function/let:
      (defun nested-function (a)
        (+ a a))
      ;; return a value for this inner let statement (that defines
'z'):
      (nested-function z)))
  ;; print a few blank lines, then test function 'test':
  (format t "~%~%test result is ~A~%~%" (test x y)))
```

The outer let form defines two local (lexically scoped) variables **x** and **y** with and assigns them the values 1 and 2 respectively. The inner function nested-function is contained inside a let statement, which is contained inside the definition of function test, which is contained inside the outer let statement that defines the local variables x and y. The

Copyright 2002, Mark Watson. All rights reserved.

format function is used for formatted I/O. If the first argument is t, then output goes to standard output. The second (also required) argument to the format function is a string containing formatting information. The **~A** is used to print the value of any Lisp variable. The format function expects a Lisp variable or expression argument for each **~A** in the formatting string. The ~%, prints a new line. Instead of using ~%~%, to print two new line characters, we could have used an abbreviation ~2%. We will cover file I/O in a later chapter and also discuss other I/O functions like **print**, **read**, **read-line**, and **princ**.

If we use the Lisp load function to evaluate the contents of the file **nested.lisp**, we see:

```
> (load "nested.lisp")
;; Loading file nested.lisp ...

test result is 6

;; Loading of file nested.lisp is finished.
T
>
```

The function load returned a value of **t** (prints in upper case as **T**) after successfully loading the file.

We will use Common Lisp vectors and arrays frequently in later chapters, but will also briefly introduce them here. A singly dimensioned array is also called a vector. Although there are often more efficient functions for handling vectors, we will just look at generic functions that handle any type of array, including vectors. Common Lisp provides support for functions with the same name that take different argument types; we will discuss this is some detail when we cover CLOS in Chapter 8. We will start by defining three vectors **v1**, **v2**, and **v3**:

```
> (setq v1 (make-array '(3)))
#(NIL NIL NIL)
> (setq v2 (make-array '(4) :initial-element "lisp is good"))
#("lisp is good" "lisp is good" "lisp is good" "lisp is good")
> (setq v3 #(1 2 3 4 "cat" '(99 100)))
#(1 2 3 4 "cat" '(99 100))
```

The function **aref** can be used to access any element in an array:

```
> (aref v3 3)
4
> (aref v3 5)
'(99 100)
>
```

Notice how indexing of arrays is zero-based; that is, indices start at zero for the first element of a sequence. Also notice that array elements can be any Lisp data type. So far, we have used the special operator **setq** to set the value of a variable. Common Lisp has a generalized version of **setq** called **setf** that can set any value in a list, array, hash table,

Copyright 2002, Mark Watson. All rights reserved.

etc. You can use **setf** instead of **setq** in all cases, but not vice-versa. Here is a simple example:

```
> v1
#(NIL NIL NIL)
> (setf (aref v1 1) "this is a test")
"this is a test"
> v1
#(NIL "this is a test" NIL)
>
```

When writing new code or doing quick programming experiments, it is often easiest (i.e., quickest to program) to use lists to build interesting data structures. However, as programs mature, it is common to modify them to use more efficient (at runtime) data structures like arrays and hash tables.

2.1 Symbols

We will discuss symbols in more detail in Chapter 7 when we cover Common Lisp Packages. For now, it is enough for you to understand that symbols can be names that refer to variables. For example:

```
> (defvar *cat* "bowser")
*CAT*
> *cat*
"bowser"
> (defvar *l* (list cat))
*L*
> *l*
("bowser")
>
```

Note that the first **defvar** returns the defined symbol as its value. Symbols are almost always converted to upper case. An exception to this "upper case rule" is when we define symbols that may contain white space using vertical bar characters:

```
> (defvar |a symbol with Space Characters| 3.14159)
|a symbol with Space Characters|
> |a symbol with Space Characters|
3.14159
>
```

2.2 Operations on Lists

Lists are a fundamental data structure of Lisp. In this section, we will look at some of the more commonly used functions that operate on lists. All of the functions described in this section have something in common: they do not modify their arguments.

Copyright 2002, Mark Watson. All rights reserved.

In Lisp, a cons cell is a data structure containing two pointers. Usually, the first pointer in a cons cell will point to the first element in a list and the second pointer will point to another cons representing the start of the rest of the original list.

The function cons takes two arguments that it stores in the two pointers of a new cons data structure. For example:

```
> (cons 1 2)
(1 . 2)
> (cons 1 '(2 3 4))
(1 2 3 4)
>
```

The first form evaluates to a cons data structure while the second evaluates to a cons data structure that is also a proper list. The difference is that in the second case the second pointer of the freshly created cons data structure points to another cons cell.

First, we will declare two global variables **l1** and **l2** that we will use in our examples. The list **l1** contains three elements and the list **l2** contains four elements:

```
> (defvar l1 '(1 2 (3 4 (5 6))))
L1
> (defvar l2 '(the "dog" calculated 3.14159))
L2
> l1
(1 2 (3 4 (5 6)))
> l2
(THE "dog" CALCULATED 3.14159)
>
```

The list referenced by the special global variable **l1** is seen in Figure 2.1 that shows the cons cells used to construct the list. You can also use the function list to create a new list; the arguments passed to function **list** are the elements of the created list:

```
> (list 1 2 3 'cat "dog")
(1 2 3 CAT "dog")
>
```

The function **car** returns the first element of a list and the function **cdr** returns a list with its first element removed (but does not modify its argument):

```
> (car l1)
1
> (cdr l1)
(2 (3 4 (5 6)))
>
```

Using combinations of **car** and **cdr** calls can be used to extract any element of a list:

```
> (car (cdr l1))
2
```

Copyright 2002, Mark Watson. All rights reserved.

```
> (cadr l1)
2
>
```

Notice that we can combine calls to **car** and **cdr** into a single function call, in this case the function **cadr**. Common Lisp defines all functions of the form **cXXr**, **cXXXr**, and **cXXXXr** where **X** can be either "a" or "d".

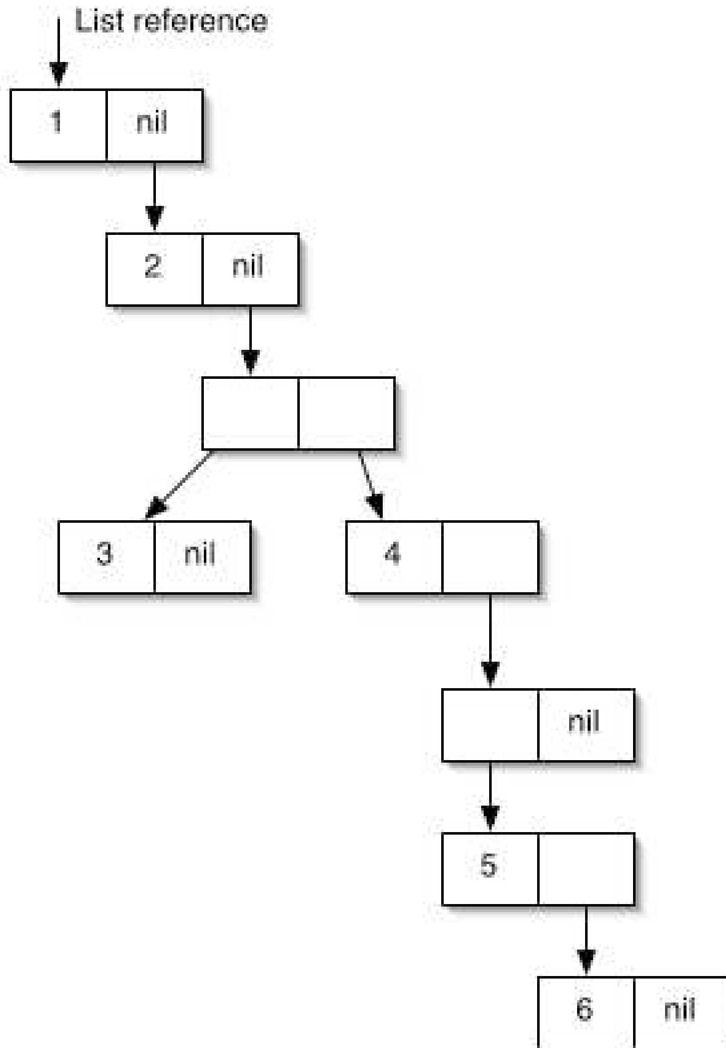

Figure 2.1: The cons cells used to construct the list '(1 2 (3 4 (5 6)))

Suppose that we want to extract the value 5 from the nested list **l1**. Some experimentation with using combinations of **car** and **cdr** gets the job done:

```
> l1
(1 2 (3 4 (5 6)))
> (cadr l1)
2
> (caddr l1)
(3 4 (5 6))
> (caddr (caddr l1))
```

Copyright 2002, Mark Watson. All rights reserved.

```
(5 6)
> (car (caddr (caddr l1)))
5
>
```

The function **last** returns the last **cdr** of a list (this is the last element, in a list):

```
> (last l1)
((3 4 (5 6)))
>
```

The function **nth** takes two arguments: an index of a top-level list element and a list. The first index argument is zero based:

```
> l1
(1 2 (3 4 (5 6)))
> (nth 0 l1)
1
> (nth 1 l1)
2
> (nth 2 l1)
(3 4 (5 6))
>
```

The function **cons** adds an element to the beginning of a list and returns as its value a new list (it does not modify its arguments). An element added to the beginning of a list can be any Lisp data type, including another list:

```
> (cons 'first l1)
(FIRST 1 2 (3 4 (5 6)))
> (cons '(1 2 3) '(11 22 33))
((1 2 3) 11 22 33)
>
```

The function **append** takes two lists as arguments and returns as its value the two lists appended together:

```
> l1
(1 2 (3 4 (5 6)))
> l2
('THE "dog" 'CALCULATED 3.14159)
> (append l1 l2)
(1 2 (3 4 (5 6)) 'THE "dog" 'CALCULATED 3.14159)
> (append '(first) l1)
(FIRST 1 2 (3 4 (5 6)))
>
```

A frequent error that beginning Lisp programmers make is not understanding shared structures in lists. Consider the following example where we generate a list **y** by reusing three copies of the list **x**:

```
> (setq x '(0 0 0 0))
(0 0 0 0)
```

Copyright 2002, Mark Watson. All rights reserved.

```
> (setq y (list x x x))
((0 0 0 0) (0 0 0 0) (0 0 0 0))
> (setf (nth 2 (nth 1 y)) 'x)
X
> x
(0 0 X 0)
> y
((0 0 X 0) (0 0 X 0) (0 0 X 0))
> (setq z '((0 0 0 0) (0 0 0 0) (0 0 0 0)))
((0 0 0 0) (0 0 0 0) (0 0 0 0))
> (setf (nth 2 (nth 1 z)) 'x)
X
> z
((0 0 0 0) (0 0 X 0) (0 0 0 0))
>
```

When we change the shared structure referenced by the variable **x** that change is reflected three times in the list **y**. When we create the list stored in the variable **z** we are not using a shared structure.

2.3 Using arrays and vectors

Using lists is easy but the time spent accessing a list element is proportional to the length of the list. Arrays and vectors are more efficient at runtime than long lists because list elements are kept on a linked-list that must be searched. Accessing any element of a short list is fast, but for sequences with thousands of elements, it is faster to use vectors and arrays.

By default, elements of arrays and vectors can be any Lisp data type. There are options when creating arrays to tell the Common Lisp compiler that a given array or vector will only contain a single data type (e.g., floating point numbers) but we will not use these options in this book.

Vectors are a specialization of arrays; vectors are arrays that only have one dimension. For efficiency, there are functions that only operate on vectors, but since array functions also work on vectors, we will concentrate on arrays. In the next section, we will look at character strings that are a specialization of vectors.

Since arrays are sequences, we could use the generalized **make-sequence** function to make a singularly dimensioned array (i.e., a vector):

```
> (defvar x (make-sequence 'vector 5 :initial-element 0))
X
> x
#(0 0 0 0 0)
>
```

In this example, notice the print format for vectors that looks like a list with a proceeding # character. We use the function **make-array** to create arrays:

Copyright 2002, Mark Watson. All rights reserved.

```
> (defvar y (make-array '(2 3) :initial-element 1))
Y
> y
#2A((1 1 1) (1 1 1))
>
```

Notice the print format of an array: it looks like a list proceeded by a # character and the integer number of dimensions.

Instead of using **make-sequence** to create vectors, we can pass an integer as the first argument of make-array instead of a list of dimension values. We can also create a vector by using the function **vector** and providing the vector contents as arguments:

```
> (make-array 10)
#(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
> (vector 1 2 3 'cat)
#(1 2 3 CAT)
>
```

The function **aref** is used to access array elements. The first argument is an array and the remaining argument(s) are array indices. For example:

```
> x
#(0 0 0 0 0)
> (aref x 2)
0
> (setf (aref x 2) "parrot")
"parrot"
> x
#(0 0 "parrot" 0 0)
> (aref x 2)
"parrot"
> (setf (aref y 1 2) 3.14159)
3.14159
> y
#2A((1 1 1) (1 1 3.14159))
> y
#2A((1 1 1) (1 1 1))
>
```

2.4 Using Strings

It is likely that even your first Lisp programs will involve the use of character strings. In this section, we will cover the basics: creating strings, concatenating strings to create new strings, CLISP for substrings in a string, and extracting substrings from longer strings. The string functions that we will look at here do not modify their arguments; rather, they return new strings as values. For efficiency, Common Lisp does include destructive string functions that do modify their arguments but we will not discuss these destructive functions here.

Copyright 2002, Mark Watson. All rights reserved.

We saw earlier that a string is a type of vector, which in turn is a type of array (which in turn is a type of sequence). A full coverage of the Common Lisp type system is outside the scope of this tutorial web book; a very good treatment of Common Lisp types is in Guy Steele's "Common Lisp, The Language" which is available both in print and for free on the web. Many of the built in functions for handling strings are actually more general because they are defined for the type sequence. The Common Lisp Hyperspec is another great free resource that you can find on the web. I suggest that you download an HTML version of Guy Steele's excellent reference book and the Common Lisp Hyperspec and keep both on your computer. If you continue using Common Lisp, eventually you will want to read all of Steele's book and use the Hyperspec for reference.

The following text was captured from input and output from a Common Lisp listener. First, we will declare two global variables **s1** and **space** that contain string values:

```
> (defvar s1 "the cat ran up the tree")
S1
> (defvar space " ")
SPACE
>
```

One of the most common operations on strings is to concatenate two or more strings into a new string:

```
> (concatenate 'string s1 space "up the tree")
"the cat ran up the tree up the tree"
>
```

Notice that the first argument of the function **concatenate** is the type of the sequence that the function should return; in this case, we want a string. Another common string operation is search for a substring:

```
> (search "ran" s1)
8
> (search "zzzz" s1)
NIL
>
```

If the search string (first argument to function **search**) is not found, function **search** returns **nil**, otherwise search returns an index into the second argument string. Function **search** takes several optional keyword arguments (see Chapter 3 for a discussion of keyword arguments):

```
  (search search-string a-longer-string :from-end :test
                                        :test-not :key
                                        :start1 :start2
                                        :end1 :end2)
```

For our discussion, we will just use the keyword argument **:start2** for specifying the starting search index in the second argument string and the **:from-end** flag to specify that

Copyright 2002, Mark Watson. All rights reserved.

search should start at the end of the second argument string and proceed backwards to the beginning of the string:

```
> (search " " s1)
3
> (search " " s1 :start2 5)
7
> (search " " s1 :from-end t)
18
>
```

The sequence function **subseq** can be used for strings to extract a substring from a longer string:

```
> (subseq s1 8)
"ran up the tree"
>
```

Here, the second argument specifies the starting index; the substring from the starting index to the end of the string is returned. An optional third index argument specifies one greater than the last character index that you want to extract:

```
> (subseq s1 8 11)
"ran"
>
```

It is frequently useful remove white space (or other) characters from the beginning or end of a string:

```
> (string-trim '(#\space #\z #\a) " a boy said pez")
"boy said pe"
>
```

The character **#\space** is the space character. There are also utility functions for making strings upper or lower case:

```
> (string-upcase "The dog bit the cat.")
"THE DOG BIT THE CAT."
> (string-downcase "The boy said WOW!")
"the boy said wow!"
>
```

We have not yet discussed equality of variables. The function **eq** returns true if two variables refer to the same data in memory. The function **eql** returns true if the arguments refer to the same data in memory or if they are equal numbers or characters. The function **equal** is more lenient: it returns true if two variables print the same when evaluated. More formally, function **equal** returns true if the **car** and **cdr** recursively equal to each other. An example will make this clearer:

```
> (defvar x '(1 2 3))
X
```

Copyright 2002, Mark Watson. All rights reserved.

```
> (defvar y '(1 2 3))
Y
> (eql x y)
NIL
> (equal x y)
T
> x
(1 2 3)
> y
(1 2 3)
>
```

For strings, the function **string=** is slightly more efficient than using the function **equal**:

```
> (eql "cat" "cat")
NIL
> (equal "cat" "cat")
T
> (string= "cat" "cat")
T
>
```

Common Lisp strings are sequences of characters. The function **char** is used to extract individual characters from a string:

```
> s1
"the cat ran up the tree"
> (char s1 0)
#\t
> (char s1 1)
#\h
>
```

2.5 Using hash tables

Hash tables are an extremely useful data type. While it is true that you can get the same effect by using lists and the **assoc** function, hash tables are much more efficient than lists if the lists contain many elements. For example:

```
> (defvar x '((1 2) ("animal" "dog")))
X
> (assoc 1 x)
(1 2)
> (assoc "animal" x)
NIL
> (assoc "animal" x :test #'equal)
("animal" "dog")
>
```

The second argument to function **assoc** is a list of cons cells. Function **assoc** searches for a sub-list (in the second argument) that has its **car** (i.e., first element) equal to the first argument to function **assoc**. The (perhaps) surprising thing about this example is that

Copyright 2002, Mark Watson. All rights reserved.

assoc seems to work with an integer as the first argument but not with a string. The reason for this is that by default the test for equality is done with **eql** that tests two variables to see if they refer to the same memory location or if they are identical if they are numbers. In the last call to **assoc** we used ":test #'equal" to make assoc use the function equal to test for equality.

The problem with using lists and **assoc** is that they are very inefficient for large lists. We will see that it is no more difficult to code with hash tables.

A hash table stores associations between key and value pairs, much like our last example using the assoc function. By default, hash tables use **eql** to test for equality when looking for a key match. We will duplicate the previous example using hash tables:

```
> (defvar h (make-hash-table))
H
> (setf (gethash 1 h) 2)
2
> (setf (gethash "animal" h) "dog")
"dog"
> (gethash 1 h)
2 ;
T
> (gethash "animal" h)
NIL ;
NIL
>
```

Notice that **gethash** returns multiple values: the first value is the value matching the key passed as the first argument to function **gethash** and the second returned value is true if the key was found and nil otherwise. The second returned value could be useful if hash values are **nil**.

Since we have not yet seen how to handle multiple returned values from a function, we will digress and do so here (there are many ways to handle multiple return values and we are just covering one of them):

```
> (multiple-value-setq (a b) (gethash 1 h))
2
> a
2
> b
T
>
```

Assuming that variables **a** and **b** are already declared, the variable **a** will be set to the first returned value from **gethash** and the variable **b** will be set to the second returned value.

If we use symbols as hash table keys, then using **eql** for testing for equality with hash table keys is fine:

Copyright 2002, Mark Watson. All rights reserved.

> (setf (gethash 'bb h) 'aa)
AA
> (gethash 'bb h)
AA ;
T
>

However, we saw that **eql** will not match keys with character string values. The function **make-hash-table** has optional key arguments and one of them will allow us to use strings as hash key values:

```
(make-hash-table &key :test :size :rehash-size :rehash-threshold)
```

Here, we are only interested in the first optional key argument **:test** that allows us to use the function **equal** to test for equality when matching hash table keys. For example:

```
> (defvar h2 (make-hash-table :test #'equal))
H2
> (setf (gethash "animal" h2) "dog")
"dog"
> (setf (gethash "parrot" h2) "Brady")
"Brady"
> (gethash "parrot" h2)
"Brady" ;
T
>
```

It is often useful to be able to enumerate all the key and value pairs in a hash table. Here is a simple example of doing this by first defining a function **my-print** that takes two arguments, a key and a value. We can then use the **maphash** function to call our new function **my-print** with every key and value pair in a hash table:

```
> (defun my-print (a-key a-value)
        (format t "key: ~A value: ~A~\%" a-key a-value))
MY-PRINT
> (maphash #'my-print h2)
key: parrot value: Brady
key: animal value: dog
NIL
>
```

There are a few other useful hash table functions that we demonstrate here:

```
> (hash-table-count h2)
2
> (remhash "animal" h2)
T
> (hash-table-count h2)
1
> (clrhash h2)
#S(HASH-TABLE EQUAL)
> (hash-table-count h2)
```

Copyright 2002, Mark Watson. All rights reserved.

```
0
>
```

The function **hash-table-count** returns the number of key and value pairs in a hash table. The function **remhash** can be used to remove a single key and value pair from a hash table. The function **clrhash** clears out a hash table by removing all key and value pairs in a hash table.

It is interesting to note that **clrhash** and **remhash** are the first Common Lisp functions that we have seen (so far) that modify any of its arguments, except for **setq** and **setf** that are macros and not functions.

2.6 Using Eval to evaluate Lisp Forms

We have seen how we can type arbitrary Lisp expressions in the Lisp listener and then they are evaluated. We will see in Chapter 6 that the Lisp function "read" evaluates lists (or forms) and indeed the Lisp listener uses function **read**.

In this section, we will use the function **eval** to evaluate arbitrary Lisp expressions inside a program. As a simple example:

```
> (defvar x '(+ 1 2 3 4 5))
X
> x
(+ 1 2 3 4 5)
> (eval x)
15
>
```

Using the function **eval**, we can build lists containing Lisp code and evaluate generated code inside our own programs. We get the effect of "data is code". A classic Lisp program, the OPS5 expert system tool, stored snippets of Lisp code in a network data structure and used the function **eval** to execute Lisp code stored in the network. A warning: the use of **eval** is likely to be inefficient. For efficiency, the OPS5 program contained its own version of **eval** that only interpreted a subset of Lisp used in the network.

2.7 Using a text editor to edit Lisp source files

I usually use Emacs, but we will briefly discuss the editor vi also. If you use vi (e.g., enter “vi nested.lisp”) the first thing that you should do is to configure vi to indicate matching opening parentheses whenever a closing parentheses is typed; you do this by typing “:set sm” after vi is running.

If you choose to learn Emacs, enter the following in your **.emacs** file (or your **_emacs** file

Copyright 2002, Mark Watson. All rights reserved.

in your home directory if you are running Windows):

```
(set-default 'auto-mode-alist
             (append '(("\\.lisp$" . lisp-mode)
                       ("\\.lsp$" . lisp-mode)
                       ("\\.cl$" . lisp-mode))
                     auto-mode-alist))
```

Now, whenever you open a file with the extension of “lisp”, “lsp”, or “cl” (for “Common Lisp”) then Emacs will automatically use a Lisp editing mode. I recommend searching the web using keywords “Emacs tutorial” to learn how to use the basic Emacs editing commands - we will not repeat this information here.

I do my professional Lisp programming using free software tools: Emacs, CLISP, and ILISP (although I distribute my commercial Lisp applications using the excellent Xanalys LispWorks product for Linux and Windows). ILISP is an excellent extension to Emacs to facilitate Lisp development. After you have used CLISP and Emacs for a while, you might consider spending an evening learning how to install ILISP and how to use it. Note: Emacs, CLISP, and ILISP are portable software tools: I run them on Linux, Mac OS X, and Windows 2000.

2.8 Recovering from Errors

When you enter forms (or expressions) in a Lisp listener, you will occasionally make a mistake and an error will be thrown. For example:

```
> (defun my-add-one (x) (+ x 1))
MY-ADD-ONE
> (my-add-one 10)
11
> (my-add-one 3.14159)
4.14159
> (my-add-one "cat")

*** - argument to + should be a number: "cat"
1. Break > :bt

EVAL frame for form (+ X 1)
APPLY frame for call (MY-ADD-ONE '"cat")
EVAL frame for form (MY-ADD-ONE "cat")

1. Break > :a

>
```

Here, I first used the **backtrace** command “:bt” to print the sequence of function calls that caused the error. If it is obvious where the error is in the code that I am working on then I do not bother using the **backtrace** command. I then used the abort command “:a” to recover back to the top level Lisp listener (i.e., back to the greater than prompt).

Copyright 2002, Mark Watson. All rights reserved.

Sometimes, you must type ":a" more than once to fully recover to the top level greater than prompt.

2.9 Garbage collection

Like other languages like Java and Python, Common Lisp provides garbage collection (GC) or automatic memory management.

In simple terms, GC occurs to free memory in a Lisp environment that is no longer accessible by any global variable (or function closure, which we will cover in the next chapter). If a global variable ***variable-1*** is first set to a list and then if we later then set ***variable-1*** to, for example nil, and if the data referenced in the original list is not referenced by any other accessible data, then this now unused data is subject to GC.

In practice, memory for Lisp data is allocated in time ordered batches and ephemeral or generational garbage collectors garbage collect recent memory allocations far more often than memory that has been allocated for a longer period of time.

2.10 Loading your Working Environment Quickly

When you start using Common Lisp for large projects, you will likely have both many files to load into your Lisp environment when you start working. Most Common Lisp implementations have a system called **defsystem** that works somewhat like the Unix **make** utility. While I strongly recommend **defsystem** for large multi-person projects, I usually use a simpler scheme when working on my own: I place a file **loadit.lisp** in the top directory of each project that I work on. For any project, its **loadit.lisp** file loads all source files and initializes any global data for the project. Since I work in Emacs with ILisp, after a project is loaded, I just need to recompile individual functions as I modify them, or I can always re-load the file **loadit.lisp** if I have changed many files.

Another good technique is to create a Lisp image containing all the code and data for all your projects. In CLisp, after you load data and Lisp code into a working image, you can evaluate **(saveinitimage)** to create a file called **lispinit.mem** in the current directory. For work on my KnowledgeBooks.com products, I need to load in a large amount of data for my work so I save a working image, change the file name to **all-data.mem**, and start CLisp using:

```
/usr/local/bin/clisp -M /Users/markw/bin/all-data.mem
```

This saves me a lot of time since it takes over a minute to load all of the data that I need into an original CLisp image but it only takes a few seconds to start CLisp using the **all-data.mem** image.

Copyright 2002, Mark Watson. All rights reserved.

All Common Lisp implementations have a mechanism for dumping a working image containing code and data.

Copyright 2002, Mark Watson. All rights reserved.

3. Defining Lisp Functions

In the last chapter, we defined a few simple functions. In this chapter, we will discuss how to write functions that take a variable number of arguments, optional arguments, and keyword arguments.

The special form **defun** is used to define new functions either in Lisp source files or at the top level Lisp listener prompt. Usually, it is most convenient to place function definitions in a source file and use the function load to load them into our Lisp working environment.

In general, it is bad form to use global variables inside Lisp functions. Rather, we prefer to pass all required data into a function via its argument list and to get the results of the function as the value (or values) returned from a function. Note that if we do require global variables, it is customary to name them with beginning and ending “*” characters; for example:

```
(defvar *lexical-hash-table*
        (make-hash-table :test #’equal :size 5000))
```

Then, in this example, if you see the variable ***lexical-hash-table*** inside a function definition, then you will know that at least by naming convention, that this is a global variable.

In Chapter 1, we already saw an example of using lexically scoped local variables lexically scoped local variables inside a function definition (in the example file **nested.lisp**).

There are several options for defining the arguments that a function can take. The fastest way to introduce the various options is with a few examples.

First, we can use the &aux keyword to declare local variables for use in a function definition:

```
> (defun test (x &aux y)
       (setq y (list x x))
       y)
TEST
> (test 'cat)
(CAT CAT)
> (test 3.14159)
(3.14159 3.14159)
```

It is considered better coding style to use the **let** special operator for defining auxiliary local variables; for example:

```
> (defun test (x)
       (let ((y (list x x)))
```

Copyright 2002, Mark Watson. All rights reserved.

```
        y))
TEST
> (test "the dog bit the cat")
("the dog bit the cat" "the dog bit the cat")
>
```

You will probably not use &aux very often, but there are two other options for specifying function arguments: &optional and &key.

The following code example shows how to use optional function arguments. Note that optional arguments must occur after required arguments.

```
> (defun test (a &optional b (c 123))
        (format t "a=~A b=~A c=~A~%" a b c))
TEST
> (test 1)
a=1 b=NIL c=123
NIL
> (test 1 2)
a=1 b=2 c=123
NIL
> (test 1 2 3)
a=1 b=2 c=3
NIL
> (test 1 2 "Italian Greyhound")
a=1 b=2 c=Italian Greyhound
NIL
>
```

In this example, the optional argument **b** was not given a default value so if unspecified it will default to **nil**. The optional argument **c** is given a default value of 123.

We have already seen the use of keyword arguments in built-in Lisp functions. Here is an example of how to specify key word arguments in your functions:

```
> (defun test (a &key b c)
        (format t "a=~A b=~A c=~A~%" a b c))
TEST
> (test 1)
a=1 b=NIL c=NIL
NIL
> (test 1 :c 3.14159)
a=1 b=NIL c=3.14159
NIL
> (test "cat" :b "dog")
a=cat b=dog c=NIL
NIL
>
```

3.1 Using lambda forms

Copyright 2002, Mark Watson. All rights reserved.

It is often useful to define unnamed functions. We can define an unnamed function using **lambda**; for example, let's look at the example file **src/lambda1.lisp**. But first, we will introduce the Common Lisp function **funcall** that takes one or more arguments; the first argument is a function and any remaining arguments are passed to the function bound to the first argument. For example:

```
> (funcall 'print 'cat)
CAT
CAT
> (funcall '+ 1 2)
3
> (funcall #'- 2 3)
-1
>
```

In the first two calls to **funcall** here, we simply quote the function name that we want to call. In the third example, we use a better notation by quoting with #'. We use the #' characters to quote a function name. Here is the example file **src/lambda1.lisp**:

```
(defun test ()
  (let ((my-func
          (lambda (x) (+ x 1))))
    (funcall my-func 1)))
```

Here, we define a function using **lambda** and set the value of the local variable **my-func** to the unnamed function's value. Here is output from the function test:

```
> (test)
2

>
```

The ability to use functions as data is surprisingly useful. For now, we will look at a simple example:

```
> (testfn #'+ 100)
101
> (testfn #'print 100)

100
100
>
```

Notice that the second call to function **testfn** prints "100" twice: the first time as a side effect of calling the function **print** and the second time as the returned value of **testfn** (the function **print** returns what it is printing as its value).

3.2 Using recursion

Copyright 2002, Mark Watson. All rights reserved.

In Chapter 5, we will see how to use special Common Lisp macros for programming repetitive loops. In this section, we will use recursion for both coding simple loops and as an effective way to solve a variety of problems that can be expressed naturally using recursion.

As usual, the example programs for this section are found in the **src** directory. In the file **src/recursion1.lisp**, we see our first example of recursion:

;; a simple loop using recursion

```
(defun recursion1 (value)
  (format t "entering recursion1(~A)~\%" value)
  (if (< value 5)
      (recursion1 (1+ value))))
```

This example is more than a little sloppy, but it is useful for discussing a few points. First, notice how the function **recursion1** calls itself with an argument value of one greater than its own input argument only if the input argument "value" is less than 5. This test keeps the function from getting in an infinite loop. Here is some sample output:

```
> (load "recursion1.lisp")
;; Loading file recursion1.lisp ...
;; Loading of file recursion1.lisp is finished.
T
> (recursion1 0)
entering recursion1(0)
entering recursion1(1)
entering recursion1(2)
entering recursion1(3)
entering recursion1(4)
entering recursion1(5)
NIL
> (recursion1 -3)
entering recursion1(-3)
entering recursion1(-2)
entering recursion1(-1)
entering recursion1(0)
entering recursion1(1)
entering recursion1(2)
entering recursion1(3)
entering recursion1(4)
entering recursion1(5)
NIL
> (recursion1 20)
entering recursion1(20)
NIL
>
```

3.3 Closures

Copyright 2002, Mark Watson. All rights reserved.

We have seen that functions can both take other functions as arguments and return new functions as values. A function that references an outer lexically scoped variable is called a **closure**. The example file **src/closure1.lisp** contains a simple example:

```
(let* ((fortunes
        '("You will become a great Lisp Programmer"
          "The force will not be with you"
          "Take time for meditation"))
       (len (length fortunes))
       (index 0))
  (defun fortune ()
    (let ((new-fortune (nth index fortunes)))
      (setq index (1+ index))
      (if (>= index len) (setq index 0))
      new-fortune)))
```

Here the function **fortune** is defined inside a **let** form. Because the local variable **fortunes** is referenced inside the function **fortune**, the variable **fortunes** exists after the **let** form is evaluated. It is important to understand that usually a local variable defined inside a let form "goes out of scope" and can no longer be referenced after the let form is evaluated.

However, in this example, there is no way to access the contents of the variable fortunes except by calling the function **fortune**. At a minimum, closures are a great way to hide variables. Here is some output from loading the **src/closure1.lisp** file and calling the function **fortune** several times:

```
> (load "closure1.lisp")
;; Loading file closure1.lisp ...
;; Loading of file closure1.lisp is finished.
T
> (fortune)
"You will become a great Lisp Programmer"
> (fortune)
"The force will not be with you"
[4]> (fortune)
"Take time for meditation"
> (fortune)
"You will become a great Lisp Programmer"
>
```

Copyright 2002, Mark Watson. All rights reserved.

4. Defining Common Lisp Macros

We saw in Chapter 2 how the Lisp function **eval** could be used to evaluate arbitrary Lisp code stored in lists. Because **eval** is inefficient, a better way to generate Lisp code automatically is to define macro expressions that are expanded inline when they are used. In most Common Lisp systems, using **eval** requires the Lisp compiler to compile a form on-the-fly which is not very efficient. Some Lisp implementations use an interpreter for **eval** which is likely to be faster but might lead to obscure bugs if the interpreter and compiled code do not function identically.

4.1 Example macro

The file **src/macro1.lisp** contains both a simple macro and a function that uses the macro:

```
;; first simple macro example:

(defmacro double-list (a-list)
  `(let ((ret nil))
    (dolist (x ,a-list)
      (setq ret (append ret (list x x))))
    ret))

;; use the macro:

(defun test (x)
  (double-list x))
```

The character **`** is used to quote a list in a special way: nothing in the list is evaluated during macro expansion unless it is immediately preceded by a comma character. In this case, we specify **,a-list** because we want the value of the macro's argument **a-list** to be substituted into the specially quoted list. We will look at **dolist** in some detail in Chapter 5 but for now it is sufficient to understand that **dolist** is used to iterate through the top-level elements of a list, for example:

```
> (dolist (x '("the" "cat" "bit" "the" "rat"))
       (print x))
"the"
"cat"
"bit"
"the"
"rat"
NIL
>
```

Returning to our macro example in the file src/macro1.lisp, we will try the function **test** that uses the macro **double-list**:

Copyright 2002, Mark Watson. All rights reserved.

```
[6]> (load "macro1.lisp")
;; Loading file macro1.lisp ...
;; Loading of file macro1.lisp is finished.
T
[7]> (test '(1 2 3))
(1 1 2 2 3 3)
[8]>
```

4.2 Using the splicing operator

Another similar example is in the file **src/macro2.lisp**:

;; another macro example that uses ,@:

```
(defmacro double-args (&rest args)
  `(let ((ret nil))
    (dolist (x ,@args)
      (setq ret (append ret (list x x))))
    ret))

;; use the macro:

(defun test (&rest x)
  (double-args x))
```

Here, the splicing operator **,@** is used to substitute in the list **args** in the macro **double-args**.

4.3 Using macroexpand-1

The function **macroexpand-1** is used to transform macros with arguments into new Lisp expressions. For example:

```
> (defmacro double (a-number)
        (list '+ a-number a-number))
DOUBLE
> (macroexpand-1 '(double n))
(+ N N) ;
T
>
```

Writing macros is an effective way to extend the Lisp language because you can control the code passed to the Common Lisp compiler. In both macro example files, when the function **test** was defined, the macro expansion is done before the compiler processes the code. We will see in the next chapter several useful macros included in Common Lisp.

We have only "scratched the surface" looking at macros; the interested reader is encouraged to search the web using, for example, "Common Lisp macros".

Copyright 2002, Mark Watson. All rights reserved.

5. Using Common Lisp Loop Macros

In this chapter, we will discuss several useful macros for performing iteration (we saw how to use recursion for iteration in Chapter 2):

dolist - a simple way to process the elements of a list
dotimes - a simple way to iterate with an integer valued loop variable
do - the most general looping macro
loop – a complex looping macro (we will only look at a few simple examples)

5.1 dolist

We saw a quick example of **dolist** in the last chapter. The arguments of the **dolist** macro are:

```
   (dolist (a-variable a-list [optional-result-value])   ...body... )
```

Usually, the **dolist** macro returns nil as its value, but we can add a third optional argument which will be returned as the generated expression's value; for example:

```
> (dolist (a '(1 2) 'done) (print a))
1
2
DONE
> (dolist (a '(1 2)) (print a))
1
2
NIL
>
```

The first argument to the **dolist** macro is a local lexically scoped variable; once the code generated by the **dolist** macro finishes executing, this variable is undefined.

5.2 dotimes

The **dotimes** macro is used when you need a loop with an integer loop index. The arguments of the **dolist** macro are:

```
   (dotimes (an-index-variable max-index-plus-one [optional-result-
value])
         ...body... )
```

Usually, the **dotimes** macro returns nil as its value, but we can add a third optional argument that will be returned as the generated expression's value; for example:

Copyright 2002, Mark Watson. All rights reserved.

```
> (dotimes (i 3 "all-done-with-test-dotimes-loop") (print i))
```

0
1
2
"all-done-with-test-dotimes-loop"
>

As with the **dolist** macro, you will often use a let form inside a **dotimes** macro to declare additional temporary (lexical) variables.

5.3 do

The **do** macro is more general purpose than either **dotimes** or **dolist** but it is more complicated to use. Here is the general form for using the do looping macro:

```
  (do ((variable-1 variable-1-init-value variable-1-update-expression)
          (variable-2 variable-2-init-value variable-2-update-
expression)
          .
          .
          (variable-N variable-N-init-value variable-N-update-
expression))
        (loop-termination-test  loop-return-value)
        optional-variable-declarations
        expressions-to-be-executed-inside-the-loop)
```

There is a similar macro **do*** that is analogous to **let*** in that loop variable values can depend on the values or previously declared loop variable values.

As a simple example, here is a loop to print out the integers from 0 to 3. This example is in the file **src/do1.lisp**:

```
;; example do macro use

(do ((i 0 (1+ i)))
        ((> i 3) "value-of-do-loop")
  (print i))
```

In this example, we only declare one loop variable so we might as well as used the simpler **dotimes** macro.

Here we load the file **src/do1.lisp**:

```
> (load "do1.lisp")
;; Loading file do1.lisp ...
0
1
```

Copyright 2002, Mark Watson. All rights reserved.

```
2
3
;; Loading of file do1.lisp is finished.
T
>
```

You will notice that we do not see the return value of the do loop (i.e., the string "value-of-do-loop") because the top-level form that we are evaluating is a call to the function **load**; we do see the return value of **load** printed. If we had manually typed this example loop in the Lisp listener, then you would see the final value **value-of-do-loop** printed.

5.4 loop

The loop macro is complex to use and we will only look at a few very simple examples of its use here. Later, in Section 9.4 we will use it when writing an email client.

<< to be done >>

Copyright 2002, Mark Watson. All rights reserved.

6. Input and Output

We will see the input and output of Lisp data is handled using streams. Streams are powerful abstractions that support common libraries of functions for writing to the terminal, to files, to sockets (covered separately in Chapter 9), and to strings.

In all cases, if an input or output function is called without specifying a stream, the default for input stream is ***standard-input*** and the default for output stream is ***standard-output***. These defaults streams are connected to the Lisp listener that we discussed in Chapter 2.

6.1 The Lisp read and read-line functions

The function **read** is used to read one Lisp expression. Function **read** stops reading after reading one expression and ignores new line characters. We will look at a simple example of reading a file **test.dat** using the example Lisp program in the file **read-test-1.lisp**. Both of these files, as usual, can be found in the directory **src** that came bundled with this web book. Start your Lisp program in the **src** directory. The contents of the file **test.dat** is:

```
1 2 3
4 "the cat bit the rat"
```

Read with-open-file

In the function **read-test-1**, we use the macro **with-open-file** to read from a file. To write to a file (which we will do later), we out use the keyword arguments **:direction :output**. The first argument to the macro **with-open-file** is a symbol that is bound to a newly created input stream (or an output stream if we are writing a file); this symbol can then be used in calling any function that expects a stream argument.

Notice that we call the function **read** with three arguments: an input stream, a flag to indicate if an error should be thrown if there is an I/O error (e.g., reaching the end of a file), and the third argument is the value that read should return if the end of the file (or stream) is reached. When calling **read** with these three arguments, either the next expression from the file **test.dat** will be returned, or the value **nil** will be returned when the end of the file is reached. If we do reach the end of the file, the local variable **x** will be assigned the value **nil** and the function **return** will break out of the **dotimes** loop. One big advantage of using the macro **with-open-file** over using the **open** function (which we will not cover) is that the file stream is automatically closed when leaving the code generated by the **with-open-file** macro. The contents of file **read-test-1.lisp** is:

```
(defun read-test-1 ()
  "read a maximum of 1000 expressions from the file 'test.dat'"
  (with-open-file
```

Copyright 2002, Mark Watson. All rights reserved.

```
   (input-stream "test.dat" :direction :input)
   (dotimes (i 1000)
     (let ((x (read input-stream nil nil)))
       (if (null x) (return)) ;; break out of the 'dotimes' loop
       (format t "next expression in file: ~S~%" x)))))
```

Here is the output that you will see if you load the file **read-test-1.lisp** and execute the expression **(read-test-1)**:

```
> (load "read-test-1.lisp")
;; Loading file read-test-1.lisp ...
;; Loading of file read-test-1.lisp is finished.
T
> (read-test-1)
next expression in file: 1
next expression in file: 2
next expression in file: 3
next expression in file: 4
next expression in file: "the cat bit the rat"
NIL
```

Note: the string "the cat bit the rat" prints as a string (with quotes) because we used a **~S** instead of a **~A** in the format string in the call to function **format**.

In this last example, we passed the file name as a string to the macro **with-open-file**. This is not in general portable across all operating systems. Instead, we could have created a pathname object and passed that instead. The pathname function can take 8 different keyword arguments, but we will use only the two most common in the example in the file **read-test-2.lisp** in the **src** directory. The following listing shows just the differences between this example and the last:

```
  (let ((a-path-name (make-pathname :directory "testdata" :name
"test.dat")))
    (with-open-file
     (input-stream a-path-name :direction :input)
```

Here, we are specifying that we should look for the for the file **test.dat** in the subdirectory **testdata**. Note: I almost never use pathnames. Instead, I specify files using a string and the character / as a directory delimiter. I find this to be portable for the Macintosh, Windows, and Linux operating systems using CLisp, OpenMCL, and LispWorks.

The file **readline-test.lisp** is identical to the file **read-test-1.lisp** except that we call function **readline** instead of the function **read** and we change the output format message to indicate that an entire line of text has been read

```
(defun readline-test ()
  "read a maximum of 1000 expressions from the file 'test.dat'"
  (with-open-file
   (input-stream "test.dat" :direction :input)
   (dotimes (i 1000)
     (let ((x (read-line input-stream nil nil)))
```

Copyright 2002, Mark Watson. All rights reserved.

```
      (if (null x) (return)) ;; break out of the 'dotimes' loop
      (format t "next line in file: ~S~%" x)))))
```

When we execute the expression **(readline-test)**, notice that the string contained in the second line of the input file has the quote characters escaped:

```
> (load "readline-test.lisp")
;; Loading file readline-test.lisp ...
;; Loading of file readline-test.lisp is finished.
T
> (readline-test)
next line in file: "1 2 3"
next line in file: "4 \"the cat bit the rat\""
NIL
>
```

We can also create an input stream from the contents of a string. The file **read-from-string-test.lisp** is very similar to the example file **read-test-1.lisp** except that we use the macro **with-input-from-string** (notice how I escaped the quote characters used inside the test string):

```
(defun read-from-string-test ()
  "read a maximum of 1000 expressions from a string"
  (let ((str "1 2 \"My parrot is named Brady.\" (11 22)"))
    (with-input-from-string
     (input-stream str)
     (dotimes (i 1000)
       (let ((x (read input-stream nil nil)))
         (if (null x) (return)) ;; break out of the 'dotimes' loop
         (format t "next expression in string: ~S~%" x))))))
```

We see the following output when we load the file **read-from-string-test.lisp**:

```
> (load "read-from-string-test.lisp")
;; Loading file read-from-string-test.lisp ...
;; Loading of file read-from-string-test.lisp is finished.
T
> (read-from-string-test)
next expression in string: 1
next expression in string: 2
next expression in string: "My parrot is named Brady."
next expression in string: (11 22)
NIL
>
```

We have seen how the stream abstraction is useful for allowing the same operations on a variety of stream data. In the next section, we will see that this generality also applies to the Lisp printing functions.

6.2 Lisp printing functions

Copyright 2002, Mark Watson. All rights reserved.

All of the printing functions that we will look at in this section take an optional last argument that is an output stream. The exception is the format function that can take a stream value as its first argument (or **t** to indicate ***standard-output*,** or a **nil** value to indicate that format should return a string value).

Here is an example of specifying the optional stream argument:

```
> (print "testing")

"testing"
"testing"
> (print "testing" *standard-output*)

"testing"
"testing"
>
```

The function **print** prints Lisp objects so that they can (usually) be read back using function read. The corresponding function **princ** is used to print for "human consumption". For example:

```
> (print "testing")

"testing"
"testing"
> (princ "testing")
testing
"testing"
>
```

Both **print** and **princ** return their first argument as their return value, which you see in the previous output. Notice that **princ** also does not print a new line character, so **princ** is often used with **terpri** (which also takes an optional stream argument).

We have also seen many examples in this web book of using the **format** function. Here is a different use of **format**, building a string by specifying the value **nil** for the first argument:

```
> (let ((l1 '(1 2))
           (x 3.14159))
       (format nil "~A~A" l1 x))
"(1 2)3.14159"
>
```

We have not yet seen an example of writing to a file. Here, we will use the **with-open-file** macro with options to write a file and to delete any existing file with the same name:

```
(with-open-file (out-stream "test1.dat"
                   :direction :output
                   :if-exists :supersede)
       (print "the cat ran down the road" out-stream)
       (format out-stream "1 + 2 is: ~A~%" (+ 1 2))
```

Copyright 2002, Mark Watson. All rights reserved.

```
        (princ "Stoking!!" out-stream)
        (terpri out-stream))
```

Here is the result of evaluating this expression (i.e., the contents of the newly created file **test1.dat**):

```
[localhost:~/Content/Loving-Lisp/src] markw% cat test1.dat

"the cat ran down the road" 1 + 2 is: 3
Stoking!!
[localhost:~/Content/Loving-Lisp/src] markw%
```

Notice that print generates a new line character before printing its argument.

Copyright 2002, Mark Watson. All rights reserved.

7. Common Lisp Package System

In the simple examples that we have seen so far, all newly created Lisp symbols have been placed in the default package. You can always check the current package by evaluating the expression ***package***:

```
> *package*
#<PACKAGE COMMON-LISP-USER>
>
```

We can always start a symbol name with a package name and two colon characters if we want to use a symbol defined in another package.

We can define new packages using **defpackage**. The following example output is long, but demonstrates the key features of using packages to partition the namespaces used to store symbols. Note: usually in this web book, I show CLISP output without the expression counter before the > character in the expression prompt; here I show the expression counter because I also want to show how CLISP prints the current package in the expression prompt if the current package is not the default COMMON-LISP-USER:

```
[1]> (defun foo1 () "foo1")
FOO1
[2]> (defpackage "MY-NEW-PACKAGE"
   (:use "COMMON-LISP-USER")
   (:nicknames "P1")
   (:export "FOO2"))
#<PACKAGE MY-NEW-PACKAGE>
[3]> (in-package my-new-package)
#<PACKAGE MY-NEW-PACKAGE>
P1[4]> (foo1)

*** - COMMON-LISP:EVAL: the function FOO1 is undefined
1. Break P1[5]> :a

P1[6]> (common-lisp-user::foo1)
"foo1"
P1[7]> (system::defun foo2 () "foo2")
FOO2
P1[8]> (system::in-package common-lisp-user)
#<PACKAGE COMMON-LISP-USER>
[9]> (foo2)

*** - EVAL: the function FOO2 is undefined
1. Break [10]> :a

[11]> (my-new-package::foo2)
"foo2"
[12]> (p1::foo2)
"foo2"
[13]>
```

Copyright 2002, Mark Watson. All rights reserved.

Since we specified a nickname in the **defpackage** expression, CLISP uses the nickname (in this case **P1** at the beginning of the expression prompt when we switch to the package **MY-NEW-PACKAGE**. Note also that we had to specify the package name when using the symbols **system::defun** and **system::in-package** to refer to these macros while in the package **MY-NEW-PACKAGE**.

Near the end of the last example, we switched back to the default package **COMMON-LISP-USER** so we had to specify the package name for the function **foo2**.

When you are writing very large Common Lisp programs, it is very useful to be able to break up the program into different modules and place each module and all its required data in different name spaces by creating new packages. Remember that all symbols, including variables, generated symbols, CLOS methods, functions, and macros are in some package.

Usually, when I use a package, I place everything in the package in a single source file. I put a **defpackage** expression at the top of the file immediately followed by an **in-package** expression to switch to the new package. Note that whenever a new file is loaded into a Lisp environment that the current package is set back to the default package **COMMON-LISP-USER**.

Since the use of packages is a common source of problems for new users, you might want to "put off" using packages until your Common Lisp programs become large enough to make the use of packages effective.

Copyright 2002, Mark Watson. All rights reserved.

8. Common Lisp Object System - CLOS

CLOS was the first ANSI standardized object oriented programming facility. While I do not use classes and objects as often in my Common Lisp programs as I do when using Java and Smalltalk, it is difficult to imagine a Common Lisp program of any size that did not define and use at least a few CLOS classes.

The example program for this chapter in the file **src/HTMLstream.lisp**. I use this CLOS class in a demo for my commercial natural language processing product to automatically generate demo web pages (see www.knowledgebooks.com if you are curious). We will also use this CLOS class in Chapter 10.

We are going to start our discussion of CLOS somewhat backwards by first looking at a short test function that uses the **HTMLstream** class. Once we understand how to use an existing class, we will introduce a small subset of CLOS by discussing in some detail the implementation of the **HTMLstream** class and finally, at the end of the chapter, see a few more CLOS programming techniques. This web book only provides a brief introduction to CLOS; the interested reader is encouraged to do a web search for "CLOS tutorial".

The macros and functions defined to implement CLOS are a standard part of Common Lisp. Common Lisp supports generic functions, that is, different functions with the same name that are distinguished by different argument types.

8.1 Example of using a CLOS class

The file **src/HTMLstream.lisp** contains a short test program at the end of the file:

```
(defun test (&aux x)
    (setq x (make-instance 'HTMLstream))
    (set-header x "test page")
    (add-element x "test text - this could be any element")
    (add-table
         x
         '(("<b>Key phrase</b>" "<b>Ranking value</b>")
           ("this is a test" 3.3)))
    (get-html-string x))
```

The generic function **make-instance** takes the following arguments:

```
    make-instance class-name &rest initial-arguments &key ...
```

There are four generic functions used in the function **test**:

set-header - required to initialize class and also defines the page title
add-element - used to insert a string that defines any type of HTML element

Copyright 2002, Mark Watson. All rights reserved.

add-table - takes a list of lists and uses the list data to construct an HTML table
get-html-string - closes the stream and returns all generated HTML data as a string

The first thing to notice in the function test is that the first argument for calling each of these generic functions is an instance of the class **HTMLstream**. You are free to also define a function, for example, add-element that does not take an instance of the class **HTMLstream** as the first function argument and calls to add-element will be routed correctly to the correct function definition.

We will see that the macro **defmethod** acts similarly to **defun** except that it also allows us to define many methods (i.e., functions for a class) with the same function name that are differentiated by different argument types and possibly different numbers of arguments.

8.2 Implementation of the HTMLstream class

The class **HTMLstream** is very simple and will serve as a reasonable introduction to CLOS programming. Later we will see more complicated class examples that use multiple inheritance. Still, this is a good example because the code is simple and the author uses this class frequently (some proof that it is useful!). The code fragments listed in this section are all contained in the file **src/HTMLstream.lisp**. We start defining a new class using the macro **defclass** that takes the following arguments:

```
defclass class-name list-of-super-classes
         list-of-slot-specifications class-specifications
```

The class definition for **HTMLstream** is fairly simple:

```
(defclass HTMLstream ()
  ((out :accessor out))
  (:documentation "Provide HTML generation services"))
```

Here, the class name is **HTMLstream**, the list of super classes is an empty list **()**, the list of slot specifications contains only one slot specification for the slot out and there is only one class specification: a documentation string. Most CLOS classes inherit from at least one super class but we will wait until the next section to see examples of inheritance. There is only one slot (or instance variable) and we define an accessor variable with the same name as the slot name (this is a personal preference of mine to name read/write accessor variables with the same name as the slot).

The method **set-header** initializes the string output stream used internally by an instance of this class. This method uses convenience macro **with-accessors** that binds a local set-able local variable to one or more class slot accessors. We will list the entire method then discuss it:

Copyright 2002, Mark Watson. All rights reserved.

```
(defmethod set-header ((ho HTMLstream) title)
  (with-accessors
     ((out out))
     ho
   (setf out (make-string-output-stream))
   (princ "<HTML><head><title>" out)
   (princ title out)
   (princ "</title></head><BODY>" out)
   (terpri out)))
```

The first interesting thing to notice about the **defmethod** is the argument list: there are two arguments **ho** and **title** but we are constraining the argument **ho** to be either a member of the class **HTMLstream** or a subclass of **HTMLstream**. Now, it makes sense that since we are passing an instance of the class **HTMLstream** to this generic function (or method – I use the terms “generic function” and “method” interchangeably) that we would want access to the slot defined for this class. The convenience macro **with-accessors** is exactly what we need to get read and write access to the slot inside a generic function (or method) for this class. In the term **((out out))**, the first **out** is local variable bound to the value of the slot named **out** for this instance **ho** of class **HTMLstream**. Inside the **with-accessors** macro, we can now use **setf** to set the slot value to a new string output stream. Note: we have not covered the Common Lisp type **string-output-stream** yet in this web book, but we will explain its use on the next page.

By the time a call to the method **set-header** (with arguments of an **HTMLstream** instance and a string title) finishes, the instance has its slot set to a new **string-output-stream** and HTML header information is written to the newly created string output stream. Note: this string output stream is now available for use by any class methods called after **set-header**.

There are several methods defined in the file **src/HTMLstream.lisp**, but we will just look at four of them: **add-H1**, **add-element**, **add-table**, and **get-html-string**. The remaining methods are very similar to **add-H1** and the reader can read the code in the source file.

As in the method **set-header**, the method **add-H1** uses the macro **with-accessors** to access the stream output stream slot as a local variable **out**. In **add-H1** we use the function **princ** that we discussed in Chapter 6 to write HTML text to the string output stream:

```
(defmethod add-H1 ((ho HTMLstream) some-text)
  (with-accessors
   ((out out))
   ho
   (princ "<H1>" out)
   (princ some-text out)
   (princ "</H1>" out)
```

Copyright 2002, Mark Watson. All rights reserved.

```
   (terpri out)))
```

The method **add-element** is very similar to **add-H1** except the string passed as the second argument element is written directly to the stream output stream slot:

```
(defmethod add-element ((ho HTMLstream) element)
  (with-accessors
      ((out out))
      ho
    (princ element out)
    (terpri out)))
```

The method **add-table** is a utility for converting a list of lists into an HTML table. The Common Lisp function **princ-to-string** is a useful utility function for writing the value of any variable to a string. The functions **string-left-trim** and **string-right-trim** are string utility functions that take two arguments: a list of characters and a string and respectively remove these characters from either the left or right side of a string. Note: another similar function that takes the same arguments is **string-trim** that removes characters from both the front (left) and end (right) of a string. All three of these functions do not modify the second string argument; they return a new string value. Here is the definition of the **add-table** method:

```
(defmethod add-table ((ho HTMLstream) table-data)
  (with-accessors
      ((out out))
      ho
    (princ "<TABLE BORDER=\"1\" WIDTH=\"100\%\"\>" out)
    (dolist (d table-data)
      (terpri out)
      (princ "  <TR>" out)
      (terpri out)
      (dolist (w d)
        (princ "    <TD>" out)
        (let ((str (princ-to-string w)))
          (setq str (string-left-trim '(#\() str))
          (setq str (string-right-trim '(#\)) str))
          (princ str out))
        (princ "</TD>" out)
        (terpri out))
      (princ "  </TR>" out)
      (terpri out))
    (princ "</TABLE>" out)
    (terpri out)))
```

The method **get-html-string** gets the string stored in the string output stream slot by using the function **get-output-stream-string**:

```
(defmethod get-html-string ((ho HTMLstream))
  (with-accessors
      ((out out))
      ho
  (princ "</BODY></HTML>" out)
  (terpri out)
```

Copyright 2002, Mark Watson. All rights reserved.

```
  (get-output-stream-string out)))
```

8.3 Other useful CLOS features

TBD

Copyright 2002, Mark Watson. All rights reserved.

9. Network Programming

Distributed computing is pervasive – look no further that the World Wide Web, Internet chat, etc. Of course, as a Lisp programmer, you will want to do at least some of your network programming in Lisp!

Unfortunately, different Common Lisp implementations tend to have different libraries for network programming. One common toolkit, CLOCC (see clocc.sourceforge.net) has a portable library for network programming that supports CLISP, LispWorks, CMU Common Lisp, etc. Since this book focuses on using CLISP, instead of using CLOCC, all the examples will be using the socket support built in to CLISP. Even if you are using a different Common Lisp implementation, I recommend that you do use CLISP for working through both this chapter and Chapter 10. Once you are comfortable doing network programming in CLISP, you should hopefully find it easy to use the socket libraries for other Common Lisp implementations or using CLOCC.

The examples in this chapter will be simple: client and server socket examples and an example for reading email from a POP3 server. In Chapter 10 we will have even more fun with an example program that retrieves Usenet news stories from a list of Usenet news groups, removes SPAM, and generates an HTML web document for reading the articles.

9.1 An introduction to sockets

We covered I/O and streams in Chapter 6. Socket programming is similar to reading and writing to local disk files but with a few complications. Network communications frequently fail so in the general case, we have to be careful to handle errors due to broken network connections busy or crashed remote servers, etc. You have experienced network failures many times: how often do you try to visit a web site, get a server not found or available error message, but 30 seconds later you can view the same web site without any problems.

We will use a common pattern in our client side socket programming examples: we use the CLISP function **socket-connect** to create a socket connection to any server and then use the macro **unwind-protect** to trap any errors and be sure that we close a socket connection when we are done with it. We will wrap calls to **socket-connect** in a function **open-socket** so that if we switch Common Lisp environments, our network programs will likely work after changing a line or two in the wrapper function **open-socket**.

*Portability note: once a socket stream is opened, the code for using the socket is portable, usually combinations of the functions **princ**, **terpri**, **format**, **read**, **read-line**, **close**, and **force-output**. For example, with LispWorks, use the function **open-tcp-stream** instead of **socket-connect**. In my projects, I like to place all operating system and*

Copyright 2002, Mark Watson. All rights reserved.

*Common Lisp implementation specific code in a special directory **system-dependent** so it usually only takes a minute or two to switch operating systems or Lisp environments.*

Server side socket programming is slightly more complicated than client side programming. In our examples, we will assume that server side code responds quickly to client requests so that it is OK to handle one client connection at a time; waiting client connection requests are queued up and handled in order or arrival. We will start in the next section with a simple server side socket example program and then present a client example program in Section 9.3.

9.2 A server example

The file **src/server.lisp** shows the implementation of a very simple socket server:

```
(defun server ()
   (let  ((a-server-socket (socket-server 8000)))
     (dotimes (i 2) ;; accept 2 connections, then quit
       (let ((connection (socket-accept a-server-socket)))
         (let ((line (read-line connection)))
           (format t "Line from client: ~A~%" line)
           ;; send something back to the server:
           (format connection "response from server~%"))
         (close connection)))
     (socket-server-close a-server-socket)))
```

This simple example reads a line of text from a test client, prints the input text, and then sends the string “response from server” back to the client. The **dotimes** loop terminates after 2 iterations for easy testing.

In a real server application, the **dotimes** loop would loop “forever” and the server would do something practical with the input text and send back something meaningful to the client. Still, this 10 line example shows how easy socket server programming can be with CLISP. Remember, the functions **socket-server**, **socket-accept**, and **socket-server-close** are specific to CLISP. Other Common Lisp systems have similar functions for handling server side sockets.

We will write a simple client for this server in Section 9.3, but assuming that we have a test client, here is the output from our server example when we run the test client two times:

```
> (load "server")
;; Loading file /Users/markw/Content/Loving-Lisp/src/server.lisp ...
;; Loading of file /Users/markw/Content/Loving-Lisp/src/server.lisp is
finished.
T
> (server)
Line from client: test string to send to server
Line from client: test string to send to server
NIL
```

Copyright 2002, Mark Watson. All rights reserved.

```
>
```

In the **server.lisp** example, we used **read-line** to read a single line from a client. Another good alternative would be to substitute **(read connection)** for **(read-line connection)** in this example so that the server would read an entire list expression and after processing return another list to the client. Note that before sending a string we would use the **princ-to-string** function to convert an arbitrary Lisp list to a string. In this case, the client example shown in the next section would also use **read** instead of **read-line**.

9.3 A client example

We saw how simple it was to write a socket-based server in Section 9.2. In this section, we will see an equally simple client example. The client program is in the file **src/client.lisp**:

```
(defun open-socket (host port)
  (socket-connect port host))

(defun client (server port a-string)
  ;; Open connection
  (let ((socket (open-socket server port)))
    (unwind-protect
        (progn
          (format socket "~A~%" a-string)
          (force-output socket)
          (let ((response (read-line socket)))
            (format t "Response from server: ~A~%" response))))
    ;; Close socket before exiting.
    (close socket)))

(defun test ()
  (client "localhost" 8000 "test string to send to server"))
```

Here, the wrapper function **open-socket** calls the CLISP specific function **socket-connect**. If you need to run this client under a different Common Lisp system, you will have to rewrite this two-line function.

Once we have created a socket we can use it like any other Common Lisp stream for both input and output. We start by using the **format** function to send a single line of text to the remote (well, in this case *localhost*) server. We then call the **read-line** function that waits for input from the remote server over the open socket connection.

We could have skipped the use of **unwind-protect** in this example but its use is good form. With **unwind-protect**, no matter what errors occur the socket connection to the server should get closed OK.

Here is the output of running the test client two times:

```
> (load "client")
```

Copyright 2002, Mark Watson. All rights reserved.

```
;; Loading file /Users/markw/Content/Loving-Lisp/src/client.lisp ...
;; Loading of file /Users/markw/Content/Loving-Lisp/src/client.lisp is
finished.
T
> (test)
Response from server: response from server
T
> (test)
Response from server: response from server
T
>
```

9.4 An Email Client

The POP3 protocol for accessing remote email servers is text based and very simple. For an overview, tutorial, and example session using POP3 look at the web site http://www.faqs.org/rfcs/rfc1939.html. The example email client **src/pop3.lisp** is fairly simple and is similar to the **src/client.lisp** example in Section 9.3 except that we have to handle a variable number of lines from a POP3 email server.

We see something new in this example: the use of the loop macro (discussed in Chapter 5) and the function **unless**. Function **unless** is like a “negative if statement”:

```
> (unless t t)
NIL
> (unless nil t)
T
>
```

If you took a few minutes to look over the POP3 specification, you will have an easier time following the example code in this section. In any case, we will list **pop3.lisp**, and then discuss the example code:

```
(defun open-socket (host port)
  (socket-connect port host))

(defun send-line (stream line)
  "Send a line of text to the socket stream, terminating it with
CR+LF."
  (princ line stream)
  (princ #\Return stream)
  (princ #\Newline stream)
  (force-output stream))

;; a string containing a new line character:
(defvar *nl* (make-string 1 :initial-element #\newline))

;; collect all input from a socket connection into a string, stopping
;; when a line from the server just contains a single period:
(defun collect-input (socket period-flag)
  (let ((ret "")
        temp)
```

Copyright 2002, Mark Watson. All rights reserved.

```
    (loop
     (let ((line (read-line socket nil nil)))
       (unless line (return))
       ;;(princ "Line: ") (princ line) (terpri)
       (if (equal line ".") (return)) ;; nntp server terminates
                                      ;; response with a period crlf
       (if (null period-flag) (return))
       (setq ret (concatenate 'string ret line *nl*))
       (if (search "-ERR" line) (return))))
    ret))

(defun send-command-print-response (stream command period-flag)
  (terpri)
  (princ "Sending: ")
  (princ command)
  (terpri)
  (send-line stream command)
  (terpri)
  (collect-input stream period-flag))

(defun test (server user passwd &aux (ret nil))
  ;; Open connection
  (let ((socket (open-socket server 110)))
    (unwind-protect
        (progn
          (send-command-print-response socket
               (concatenate 'string "USER " user) nil)
          (send-command-print-response socket
               (concatenate 'string "PASS " passwd) nil)
          (send-command-print-response socket "STAT" nil)
          (let* ((response
                   (send-command-print-response socket "LIST" t))
                  (index1 (search "+OK " response)))
            ;;(print (list "**** response = " response
            ;;             "  index1 = " index1))

            ;; the string between index1 and index2 will contain the
            ;; number of email messages available to be read:
            (if index1
                (let ((index2
                        (search " " response :start2 (+ index1 4))))
                  ;;(print (list "**** index2 = " index2))
                  (if index2
                      (let ((count
                              (read-from-string
                                (subseq response (+ index1 4)
index2))))
                        ;;(print (list "**** count = " count))
                        (dotimes (i count)
                          (setq ret
                                (cons
                                 (send-command-print-response
                                  socket
                                  (concatenate 'string "RETR "
                                      (princ-to-string (1+ i))) t)
                                 ret))
                          ;; uncomment the following 2 lines if you
                          ;; want to delete the messages on the server:
```

Copyright 2002, Mark Watson. All rights reserved.

```
                        ;;(send-command-print-response socket
                        ;;                              "DELE 1" nil)
                        ))))))
          (send-command-print-response socket "QUIT" nil))
      ;; Close socket before exiting.
      (close socket)))
  (reverse ret))
```

The POP3 utility code is fairly simple because the POP3 protocol is simple and text based. If you are interested in automating email processing (hopefully, not to send SPAM to people!), you should start by reading the POP3 specification and try to manually *telnet* to your email server and manually check for email messages. The simple code in **pop3.lisp** automates this process.

Here is some sample output from running the POP3 test function (Note that I removed my email server name, username, and password from the following text):

```
> (test "EMAIL_SERVER.com" "USER_NAME" "A_PASSWORD")

Sending: USER USER_NAME

Line: +OK NGPopper vEL_4_16 at EMAIL_SERVER.com ready
<1120.1026120886@hawk>

Sending: PASS A_PASSWORD

Line: +OK

Sending: STAT

Line: +OK USER_NAME has 1 messages (1010 octets).

Sending: LIST

Line: +OK 1 1010
Line: +OK
Line: 1 1010
Line: .

("**** count = " 1)
Sending: RETR 1

Line: +OK 1010 octets
Line: Status:  U
Line: Return-Path: <markw@markwatson.com>
Line: Received: from aaaaa.com ([121.22.44.112])
Line:   for markw@markwatson.com; Thu, 11 Jul 2002 11:08:00 -0700
Line: User-Agent: Microsoft-Entourage/10.0.0.1309
Line: Date: Thu, 11 Jul 2002 11:08:03 -0700
Line: Subject: Test
Line: From: Mark Watson <markw@markwatson.com>
Line: To: Mark Watson <markw@markwatson.com>
Line: Message-ID: <B9531793.AF8%markw@markwatson.com>
Line: Mime-version: 1.0
```

Copyright 2002, Mark Watson. All rights reserved.

```
Line: Content-type: text/plain; charset="US-ASCII"
Line: Content-transfer-encoding: 7bit
Line:
Line: A test message
Line:
Line:
Line:
Line: .

Sending: QUIT

Line: +OK
("+OK 1010 octets
Status:  U
Return-Path: <markw@markwatson.com>
User-Agent: Microsoft-Entourage/10.0.0.1309
Date: Thu, 11 Jul 2002 11:08:03 -0700
Subject: Test
From: Mark Watson <markw@markwatson.com>
To: Mark Watson <markw@markwatson.com>
Message-ID: <B9531793.AF8%markw@markwatson.com>
Mime-version: 1.0
Content-type: text/plain; charset=\"US-ASCII\"
Content-transfer-encoding: 7bit

A test message



")
>
```

Here, I only had one short email message. The function **test** returns a list of strings, one string for each fetched email message. To process each email message, you could use the **read-from-string** macro (see Section 6.1) and the **readline** function to easily get each line of an email message.

Copyright 2002, Mark Watson. All rights reserved.

Index

&

&aux, 30
&key, 31
&optional, 31

A

accessor variable, 48
append (function), 18
aref (function), 20
arrays, 19
assoc (function), 23

C

C++, 5
car (function), 16
cdr (function), 16
char (function), 23
CLISP, 8
CLOS, 47
closure, 34
clrhash (function), 26
concatenate (function), 21
cons, 16

D

defclass, 48
defmethod, 48
defpackage, 45
defun (defines functions), 30
defvar, 9, 11, 15
do (loop macro), 38
dolist (loop macro), 37
dotimes (loop macro), 37

E

Emacs, 26
eq (function), 22
eql (function), 22
equal (function), 22
error recovery, 27
eval (function), 26

F

format (function), 43
funcall (function), 32
functions as data, 32

G

garbage collection, 28
gethash (function), 24
global variables, 11, 30

H

hash tables, 23
hash-table-count (function), 26

I

ILISP, 8
in-package, 46
input stream, 40

J

Java, 5, 6

L

lambda (special form), 32
last (function), 18
list (function), 16
lists, 15
loop macro, 39
loop macros. *See* dolist, dotimes, do

M

macro, 35
macroexpand-1 (function), 36
make-array, 19
make-hash-table (function), 25
make-instance (function), 47
make-sequence, 19
maphash (function), 25

N

natural language processing, 7
NLP, 7
nth (function), 18

O

output stream, 43

P

package, 45

Copyright 2002, Mark Watson. All rights reserved.

pathname, 41
princ (function), 43
print (function), 43
Prolog, 5

R

read (function), 40
readline (function), 41
recursion, 33
remhash (function), 26

S

search (function), 21
setf, 15
setq, 9
shared structure, 19
slot, 48
splicing operator, 36
streams, 40
string= (function), 23
strings, 20
subseq (function), 22
symbols, 15

T

terpri (function), 43
top-level Lisp prompt, 11
typep, 12

V

vi, 26

W

with-accessors (macro), 48
with-input-from-string, 42
with-open-file (macro), 43

X

Xanalys LispWorks, 27

Copyright 2002, Mark Watson. All rights reserved.