

Loving Common Lisp, or the Savvy Programmer's Secret Weapon

by Mark Watson



Loving Common Lisp, or the Savvy Programmer's Secret Weapon

Mark Watson

This book is for sale at <http://leanpub.com/lovinglisp>

This version was published on 2013-08-07



*

This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

*

© 2013 Mark Watson

TABLE OF CONTENTS

[Preface](#)

[Acknowledgments](#)

[Introduction](#)

[Why Did I Write this Book?](#)

[Free Software Tools for Common Lisp Programming](#)

[How is Lisp Different from Languages like Java and C++?](#)

[Advantages of Working in a Lisp Environment](#)

[Common Lisp Basics](#)

[Getting Started with SBCL](#)

[Making the repl Nicer using rlwrap](#)

[The Basics of Lisp Programming](#)

[Symbols](#)

[Operations on Lists](#)

[Using Arrays and Vectors](#)

[Using Strings](#)

[Using Hash Tables](#)

[Using Eval to Evaluate Lisp Forms](#)

[Using a Text Editor to Edit Lisp Source Files](#)

[Recovering from Errors](#)

[Garbage Collection](#)

[Loading your Working Environment Quickly](#)

[Functional Programming Concepts](#)

[Defining Lisp Functions](#)

[Using Lambda Forms](#)

[Using Recursion](#)

[Closures](#)

[Quicklisp](#)

[Using Quicklisp to Find Packages](#)

[Using Quicklisp to Configure Emacs and Slime](#)

[Defining Common Lisp Macros](#)

[Example Macro](#)

[Using the Splicing Operator](#)

[Using macroexpand-1](#)

[Using Common Lisp Loop Macros](#)

[dolist](#)

[dotimes](#)

[do](#)

[Common Lisp Package System](#)

[Input and Output](#)

[The Lisp read and read-line Functions](#)

[Lisp Printing Functions](#)

[Common Lisp Object System - CLOS](#)

[Example of Using a CLOS Class](#)

[Implementation of the HTMLstream Class](#)

[Network Programming](#)

[An introduction to Drakma](#)

[An introduction to Hunchentoot](#)

[Complete REST Client Server Example Using JSON for Data Serialization](#)

[Accessing Relational Databases](#)

[Using MongoDB, CouchDB, and Solr NoSQL Data Stores](#)

[MongoDB](#)

[CouchDB](#)

[A Common Lisp Solr Client](#)

[NoSQL Wrapup](#)

[Natural Language Processing](#)

[Loading and Running the NLP Library](#)

[Part of Speech Tagging](#)

[Categorizing Text](#)

[Stemming Text](#)

[Detecting People's Names and Place Names](#)

[Summarizing Text](#)
[Text Mining](#)

[Information Gathering](#)

[DBPedia Lookup Service](#)

[Web Spiders](#)

[Using Apache Nutch](#)

[Wrap Up](#)

Preface

Why Common Lisp? Isn't Common Lisp an old language? Do many people still use Common Lisp?

I believe that using Lisp languages like Common Lisp, Clojure, Racket, and Scheme are all secret weapons useful in agile software development. An interactive development process and live production updates feel like a breath of fresh air if you have development on *heavy weight* like Java Enterprise Edition (JEE).

Yes, Common Lisp is an old language but with age comes stability and extremely good compiler technology. There is also a little inconsistency between different Common Lisp systems in such things as handling threads but with a little up front knowledge you can choose which Common Lisp systems will support your requirements.

Acknowledgments

I would like to thank the following people who made suggestions for improving previous editions of this book: Sam Steingold, Andrew Philpot, Kenny Tilton, Mathew Villeneuve, Eli Draluk, Erik Winkels, Adam Shimali, and Paolo Amoroso.

I would like to also thank several people who pointed out typo errors in this book and for specific suggestions: and Martin Lightheart, Tong-Kiat Tan, and Rainer Joswig.

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.

I would especially like to thank my wife Carol Watson for her fine work in editing this book.

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.

The Common Lisp program examples are distributed [on the github repo for this book](#).

Why Did I Write this Book?

Why the title “Loving Common Lisp”? Simple! I have been using Lisp for over 30 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 both Ruby, Java and Javascript 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 I wrote the second edition of this book, I had been using Common Lisp almost exclusively for an artificial intelligence project for a health care company and for commercial product development. While working on the third edition of this book, I was not using Common Lisp professionally but since the release of the Quicklisp Common Lisp package manager I have found myself enjoying using Common Lisp more for small side projects. I use Quicklisp throughout in the new third edition example code so you can easily install required libraries.

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 not to settle for 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 or other languages besides Lisp, that even though I may get a lot of pleasure

or other languages besides Lisp, that even though I may get a lot of pleasure from the job I am not following my bliss.

My goal in this book is to introduce you to one of my favorite programming languages, 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.

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
- Clozure Common Lisp – open source with good Mac OS X and Linux support
- CMU Common Lisp – open source implementation
- SBCL – derived from CMU Common Lisp
- ECL – compiles using a separate C/C++ compiler
- ABCL – Armed Bear Common Lisp for the JVM

There are also fine commercial Common Lisp products:

- LispWorks – high quality and reasonably priced system for Windows and Linux. No charge for distributing compiled applications lispworks.com
- Allegro Common Lisp - high quality, great support and higher cost. franz.com
- MCL – Macintosh Common Lisp. I used this Lisp environment in the late 1980s. MCL was so good that I gave away my Xerox 1108 Lisp Machine and switched to a Mac and MCL for my development work. Now open source but only runs on the old MacOS

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

How is Lisp Different from Languages like Java and C++?

This is a trick question! Lisp is slightly more similar to Java than C++ because of automated memory management 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:

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

Here, in Java, variables are strongly typed so a variable x of type Float can not legally be assigned a string value: the code in line 5 would generate a compilation error.

Java and Lisp both provide 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, is no longer referenced.

Common Lisp is an ANSI standard language. Portability between different Common Lisp implementations and on different platforms is very good. I use Clozure Common Lisp, SBCL, Franz Lisp, LispWorks, and Clisp that all run well on Windows, Mac OS X, and Linux. As a Common Lisp developer you will have great flexibility in tools and platforms.

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.

In C++ programs, a common bug that affects a program's efficiency is forgetting to free memory that is no longer used. In a virtual 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 or runtime libraries manage memory for you. Languages like Lisp, Ruby, 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) as 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 and other Lisp languages like Clojure and Scheme to be about twice as efficient (again, in terms of my time) as Java. That is correct: I am claiming a four times increase in my programming productivity when using Common Lisp vs. C++.

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

Advantages of Working in a Lisp Environment

We will soon see 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 the [Chapter on Natural Language Processing](#), but let me share a use this general case for work that I do that is far more efficient in Lisp:

Much of my Lisp programming used to be writing commercial natural language processing (NLP) programs for my company www.knowledgebooks.com. My Lisp NLP code uses a huge 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 my 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. When I start working in any Common Lisp environment, 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 errors 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.

Common Lisp Basics

Getting Started with SBCL

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, slime, and SBCL. Emacs is a fine text editor that is extensible to work well with many programming languages and document types (e.g., HTML and XML). Slime is a Emacs extension package that greatly facilitates Lisp development. SBCL is a robust Common Lisp compiler and runtime system.

We will cover the Quicklisp package manager and using Quicklisp to setup Slime and Emacs in the next chapter. For now, it is sufficient for you to [download and install SBCL](#). Please install SBCL right now, if you have not already done so.

I will not spend much time covering the use of Emacs as a text editor in this book. If you already use Emacs or do not mind spending the effort to learn Emacs, then search the web first for an Emacs tutorial.

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

When we start SBCL, we see a introductory message and then an input prompt. We will start with a short tutorial, walking you through a session using SBCL repl (other Common LISP systems are very similar). A repl is an interactive console where you type expressions and see the results of evaluating these expressions. An expression can be a large block of code pasted into the repl, using the **load** function to load Lisp code into the repl, calling functions to test them, *etc.* Assuming that SBCL is installed on your system, start SBCL by running the SBCL program:

```
1 % sbcl
2 (running SBCL from: /Users/markw/sbcl)
3 This is SBCL 1.1.8.0-19cda10, an implementation of ANSI Common
```

Lisp.

4 More information about SBCL is available at
<<http://www.sbcl.org/>>.

5

6 SBCL is free software, provided as is, with absolutely no
warranty.

7 It is mostly in the public domain; some portions are provided
under

8 BSD-style licenses. See the CREDITS and COPYING files in the
9 distribution **for** more information.

10

11 * (defvar x 1.0)

12

13 X

14 * x

15

16 1.0

17 * (+ x 1)

18

19 2.0

20 * x

21

22 1.0

23 * (setq x (+ x 1))

24

25 2.0

26 * x

27

28 2.0

29 * (setq x "the dog chased the cat")

30

31 "the dog chased the cat"

32 * x

33

34 "the dog chased the cat"

35 * (quit)

We started by defining a new variable x in line 11. 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** in line 17, using the **setq** function to change the value of **x** in lines 23 and 29 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.

I learned to program Lisp in 1976 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.

Before you proceed to the next chapter, please take the time to install SBCL 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:

```
1 * (+ 1 2 3 4)
2
3 10
4 * (quit)
5 Bye.
```

Making the repl Nicer using rlwrap

While reading the last section you (hopefully!) played with the SBCL interactive repl. If you haven't played with the repl, I won't get too judgmental except to say that if you do not play with the book examples as you read you will not get the full benefit from this book.

Did you notice that the backspace key does not work in the SBCL repl? The way to fix this is to install the GNU **rlwrap** utility. On OS X, assuming that you have

[homebrew](#) installed, install rlwrap with:

```
1 brew install rlwrap
```

If you are running Ubuntu Linux, install rlwrap using:

```
1 sudo apt-get install rlwrap
```

You can then create an alias for bash or zsh using something like the following to define a command **rsbcl**:

```
1 alias rsbcl='rlwrap sbcl'
```

This is fine, just remember to run **sbcl** if you don't need rlwrap command line editing or run **rsbcl** when you do need command line editing. That said, I find that I *always* want to run SBCL with command line editing, so I redefine *sbcl* on my computers using:

```
1 -> ~ which sbcl
2 /Users/markw/sbcl/sbcl
3 -> ~ alias sbcl='rlwrap /Users/markw/sbcl/sbcl'
```

This alias is different on my laptops and servers, since I don't usually install SBCL in the default installation directory. For each of my computers, I add an appropriate alias in my .zshrc file (if I am running zsh) or my .bashrc file (if I am running bash).

The Basics of Lisp Programming

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

```
1 * 1
2 1
3 * 3.14159
4 3.14159
5 * "the dog bit the cat"
```

```

6 "the dog bit the cat"
7 * (defun my-add-one (x)
8 (+ x 1))
9 MY-ADD-ONE
10 * (my-add-one -10)
11 -9

```

Notice that when we defined the function `my-add-one` in lines 7 and 8, we split the definition over two lines and on line 8 you don't see the "*" prompt from SBCL – this lets you know that you have not yet entered a complete expression. 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 and an expression is complete when the parentheses match. I tend to count in my head, adding one for every opening parentheses and subtracting one for every closing parentheses – when I get back down to zero then the expression is complete. 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 * (+ 1 2)
2 3
3 * '(+ 1 2)
4 (+ 1 2)
5 * (quote (+ 1 2))
6 (+ 1 2)
7 *

```

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

```

1 * (defvar *x* "cat")
2 *x*
3 * *x*
4 "cat"
5 * (setq *x* "dog")

```

```

6  "dog"
7  *  *x*
8  "dog"
9  *  (setq *x* 3.14159)
10 3.14159
11 *  *x*
12 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 scoping 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):

```

1  t    [top level type (all other types are a sub-type)]
2      sequence
3          list
4          array
5              vector
6                  string
7      number
8          float
9          rational
10             integer

```

```

11          ratio
12      complex
13  character
14  symbol
15  structure
16  function
17  hash-table

```

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

```

1 * (setq x '(1 2 3))
2 (1 2 3)
3 * (typep x 'list)
4 T
5 * (typep x 'sequence)
6 T
7 * (typep x 'number)
8 NIL
9 * (typep (+ 1 2 3) 'number)
10 T
11 *

```

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

```

1 * (+ 1 2 3 4 5)
2 15
3 * *
4 15
5 * (setq x *)
6 15
7 * x
8 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

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 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):

```
1 (let ((x 1)
2       (y 2))
3   ;; define a test function nested inside a
4   ;; let statement:
5   (defun test (a b)
6     (let ((z (+ a b)))
7       ;; define a helper function nested
8       ;; inside a let/function/let:
9       (defun nested-function (a)
10        (+ a a))
11       ;; return a value for this inner let
12       ;; statement (that defines 'z'):
13       (nested-function z)))
14   ;; print a few blank lines, then test function 'test':
15   (format t "%test result is ~A%" (test x y)))
```

In lines 1 and 2 the outer let form defines two local (lexically scoped) variables **x** and **y** and assigns them the values 1 and 2 respectively. The inner function **nested-function** defined in lines 9 and 10 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 **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**.*

Assuming that we started SBCL in the **src** directory we can then use the Lisp load function to evaluate the contents of the file **nested.lisp** using the **load** function:

```
1 * (load "nested.lisp")
2 ;; Loading file nested.lisp ...
3
4 test result is 6
5
6 ;; Loading of file nested.lisp is finished.
7 T
8 *
```

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 in some detail when we cover this in the later chapter on CLOS. We will start by defining three vectors **v1**, **v2**, and **v3**:

```
1 * (setq v1 (make-array '(3)))
2 #(NIL NIL NIL)
3 * (setq v2 (make-array '(4) :initial-element "lisp is good"))
4 #("lisp is good" "lisp is good" "lisp is good" "lisp is good")
5 * (setq v3 #(1 2 3 4 "cat" '(99 100)))
6 #(1 2 3 4 "cat" '(99 100))
```

In line 1, we are defining a one-dimensional array, or vector, with three

elements. In line 3 we specify the default value assigned to each element of the array `v2`. In line 5 I use the form for specifying array literals using the special character `#`. The function **`aref`** can be used to access any element in an array:

```
1 * (aref v3 3)
2 4
3 * (aref v3 5)
4 '(99 100)
5 *
```

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

```
1 * v1
2 #(NIL NIL NIL)
3 * (setf (aref v1 1) "this is a test")
4 "this is a test"
5 * v1
6 #(NIL "this is a test" NIL)
7 *
```

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.

Symbols

We will discuss symbols in more detail the [Chapter on Common Lisp Packages](#). For now, it is enough for you to understand that symbols can be names that refer to variables. For example:

```
1 > (defvar *cat* "browser")
2 *CAT*
3 * *cat*
```

```

4 "browser"
5 * (defvar *l* (list *cat*))
6 *L*
7 * *l*
8 ("browser")
9 *

```

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:

```

1 * (defvar |a symbol with Space Characters| 3.14159)
2 |a symbol with Space Characters|
3 * |a symbol with Space Characters|
4 3.14159
5 *

```

Operations on Lists

Lists are a fundamental data structure of Common 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.

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:

```

1 * (cons 1 2)
2 (1 . 2)
3 * (cons 1 '(2 3 4))
4 (1 2 3 4)
5 *

```

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 five elements and the list **l2** contains four elements:

```
1 * (defvar l1 '(1 2 (3) 4 (5 6)))
2 l1
3 * (length l1)
4
5 5
6 * (defvar l2 '(the "dog" calculated 3.14159))
7 l2
8 * l1
9 (1 2 (3) 4 (5 6))
10 * l2
11 (THE "dog" CALCULATED 3.14159)
12 >
```

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:

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

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):

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


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

```
1 * (car (cdr l1))
2 2
3 * (cadr l1)
4 2
5 *
```

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**.

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:

```
1 * l1
2 (1 2 (3) 4 (5 6))
3 * (cadr l1)
4 2
5 * (caddr l1)
6 (3)
7 (car (caddr l1))
8 3
9 * (caar (last l1))
10 5
11 * (caar (cddddr l1))
12
13 5
14 *
```

The function **last** returns the last **cdr** of a list (i.e., the last element, in a list):

```
1 * (last l1)
2 ((5 6))
3 *
```

The function **nth** takes two arguments: an index of a top-level list element and a

list. The first index argument is zero based:

```
1 * l1
2 (1 2 (3) 4 (5 6))
3 * (nth 0 l1)
4 1
5 * (nth 1 l1)
6 2
7 * (nth 2 l1)
8 (3)
9 *
```

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:

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

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

```
1 * l1
2 (1 2 (3) 4 (5 6))
3 * l2
4 ('THE "dog" 'CALCULATED 3.14159)
5 * (append l1 l2)
6 (1 2 (3) 4 (5 6) THE "dog" CALCULATED 3.14159)
7 * (append '(first) l1)
8 (FIRST 1 2 (3) 4 (5 6))
9 *
```

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:

```

1 * (setq x '(0 0 0 0))
2 (0 0 0 0)
3 * (setq y (list x x x))
4 ((0 0 0 0) (0 0 0 0) (0 0 0 0))
5 * (setf (nth 2 (nth 1 y)) 'x)
6 X
7 * x
8 (0 0 X 0)
9 * y
10 ((0 0 X 0) (0 0 X 0) (0 0 X 0))
11 * (setq z '((0 0 0 0) (0 0 0 0) (0 0 0 0)))
12 ((0 0 0 0) (0 0 0 0) (0 0 0 0))
13 * (setf (nth 2 (nth 1 z)) 'x)
14 X
15 * z
16 ((0 0 0 0) (0 0 X 0) (0 0 0 0))
17 *

```

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.

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.

We could use the generalized **make-sequence** function to make a singularly dimensioned array (i.e., a vector). Restart sbcl and try:

```
1 * (defvar x (make-sequence 'vector 5 :initial-element 0))
2 X
3 * x
4 #(0 0 0 0 0)
5 *
```

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

```
1 * (defvar y (make-array '(2 3) :initial-element 1))
2 Y
3 * y
4 #2A((1 1 1) (1 1 1))
5 >
```

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:

```
1 * (make-array 10)
2 #(NIL NIL NIL NIL NIL NIL NIL NIL NIL NIL)
3 * (vector 1 2 3 'cat)
4 #(1 2 3 CAT)
5 *
```

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

```
1 * x
2 #(0 0 0 0 0)
```

```

3 * (aref x 2)
4 0
5 * (setf (aref x 2) "parrot")
6 "parrot"
7 * x
8 #(0 0 "parrot" 0 0)
9 * (aref x 2)
10 "parrot"
11 * y
12 #2A((1 1 1) (1 1 1))
13 * (setf (aref y 1 2) 3.14159)
14 3.14159
15 * y
16 #2A((1 1 1) (1 1 3.14159))
17 *

```

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, 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.

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 introduction to Common Lisp; 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 repl. First, we will declare two global variables **s1** and **space** that contain string values:

```
1 * (defvar s1 "the cat ran up the tree")
2 S1
3 * (defvar space " ")
4 SPACE
5 *
```

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

```
1 * (concatenate 'string s1 space "up the tree")
2 "the cat ran up the tree up the tree"
3 *
```

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:

```
1 * (search "ran" s1)
2 8
3 * (search "zzzz" s1)
4 NIL
5 *
```

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 the next chapter for a discussion of keyword arguments):

```
1 (search search-string a-longer-string :from-end :test
2                                           :test-not :key
3                                           :start1 :start2
4                                           :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 search should start at the end of the second argument string and proceed backwards to the beginning of the string:

```
1 * (search " " s1)
2 3
3 * (search " " s1 :start2 5)
4 7
5 * (search " " s1 :from-end t)
6 18
7 *
```

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

```
1 * (subseq s1 8)
2 "ran up the tree"
3 >
```

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:

```
1 * (subseq s1 8 11)
2 "ran"
3 *
```

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

```
1 * (string-trim '(#\space #\z #\a) " a boy said pez")
2 "boy said pe"
3 *
```

The character `#\space` is the space character. Other common characters that are trimmed are `#\tab` and `#\newline`. There are also utility functions for making strings upper or lower case:

```

1 * (string-upcase "The dog bit the cat.")
2 "THE DOG BIT THE CAT."
3 * (string-downcase "The boy said WOW!")
4 "the boy said wow!"
5 >

```

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 **eq1** 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:

```

1 * (defvar x '(1 2 3))
2 X
3 * (defvar y '(1 2 3))
4 Y
5 * (eq1 x y)
6 NIL
7 * (equal x y)
8 T
9 * x
10 (1 2 3)
11 * y
12 (1 2 3)
13 *

```

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

```

1 * (eq1 "cat" "cat")
2 NIL
3 * (equal "cat" "cat")
4 T
5 * (string= "cat" "cat")
6 T
7 *

```

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

```
1 * s1
2 "the cat ran up the tree"
3 * (char s1 0)
4 #\t
5 * (char s1 1)
6 #\h
7 *
```

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:

```
1 * (defvar x '((1 2) ("animal" "dog")))
2 X
3 * (assoc 1 x)
4 (1 2)
5 * (assoc "animal" x)
6 NIL
7 * (assoc "animal" x :test #'equal)
8 ("animal" "dog")
9 *
```

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 **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:

```
1 * (defvar h (make-hash-table))
2 H
3 * (setf (gethash 1 h) 2)
4 2
5 * (setf (gethash "animal" h) "dog")
6 "dog"
7 * (gethash 1 h)
8 2 ;
9 T
10 * (gethash "animal" h)
11 NIL ;
12 NIL
13 *
```

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):

```
1 * (multiple-value-setq (a b) (gethash 1 h))
2 2
3 * a
4 2
5 * b
6 T
7 *
```

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 **eq** for testing for equality with hash table keys is fine:

```
1 * (setf (gethash 'bb h) 'aa)
2 AA
3 * (gethash 'bb h)
4 AA ;
5 T
6 *
```

However, we saw that **eq** 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:

```
1 (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:

```
1 * (defvar h2 (make-hash-table :test #'equal))
2 H2
3 * (setf (gethash "animal" h2) "dog")
4 "dog"
5 * (setf (gethash "parrot" h2) "Brady")
6 "Brady"
7 * (gethash "parrot" h2)
8 "Brady" ;
9 T
10 *
```

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:

```
1 * (defun my-print (a-key a-value)
2     (format t "key: ~A value: A%" a-key a-value))
3 MY-PRINT
4 * (maphash #'my-print h2)
5 key: parrot value: Brady
6 key: animal value: dog
7 NIL
8 *
```

The function **my-print** is applied to each key/value pair in the hash table. There are a few other useful hash table functions that we demonstrate here:

```
1 * (hash-table-count h2)
2 2
3 * (remhash "animal" h2)
4 T
5 * (hash-table-count h2)
6 1
7 * (clrhash h2)
8 #S(HASH-TABLE EQUAL)
9 * (hash-table-count h2)
10 0
11 *
```

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.

Using Eval to Evaluate Lisp Forms

We have seen how we can type arbitrary Lisp expressions in the Lisp repl listener and then they are evaluated. We will see in the [Chapter on Input and](#)

[Output](#) that the Lisp function **read** evaluates lists (or forms) and indeed the Lisp repl uses function **read**.

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

```
1 * (defvar x '(+ 1 2 3 4 5))
2 X
3 * x
4 (+ 1 2 3 4 5)
5 * (eval x)
6 15
7 *
```

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 in non-compiled code. For efficiency, the OPS5 program contained its own version of **eval** that only interpreted a subset of Lisp used in the network.

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 in your home directory if you are running Windows):

```
1 (set-default 'auto-mode-alist
2       (append '(("\\.lisp$" . lisp-mode)
3               ("\\.lsp$" . lisp-mode)
4               ("\\.cl$" . lisp-mode)))
5       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

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, SBCL, Clozure Common Lisp, and Clojure. I will show you how to configure Emacs and Slime in the last section of the [Chapter on Quicklisp](#).

Recovering from Errors

When you enter forms (or expressions) in a Lisp repl listener, you will occasionally make a mistake and an error will be thrown. Here is an example where I am not showing all of the output when entering **help** when an error is thrown:

```
1 * (defun my-add-one (x) (+ x 1))
2
3 MY-ADD-ONE
4 * (my-add-one 10)
5
6 11
7 * (my-add-one 3.14159)
8
9 4.14159
10 * (my-add-one "cat")
11
12 debugger invoked on a SIMPLE-TYPE-ERROR: Argument X is not a
NUMBER: "cat"
13
14 Type HELP for debugger help, or (SB-EXT:EXIT) to exit from
SBCL.
15
16 restarts (invokable by number or by possibly-abbreviated name):
17 0: [ABORT] Exit debugger, returning to top level.
18
19 (SB-KERNEL:TWO-ARG-+ "cat" 1)
20 0] help
21
22 The debug prompt is square brackets, with number(s) indicating
```

```

the current
23   control stack level and, if you've entered the debugger
recursively, how
24   deeply recursed you are.
25
26   ...
27
28   Getting in and out of the debugger:
29   TOPLEVEL, TOP   exits debugger and returns to top level REPL
30   RESTART         invokes restart numbered as shown (prompt if not
given).
31   ERROR           prints the error condition and restart cases.
32
33   ...
34
35   Inspecting frames:
36   BACKTRACE [n]   shows n frames going down the stack.
37   LIST-LOCALS, L lists locals in current frame.
38   PRINT, P        displays function call for current frame.
39   SOURCE [n]      displays frame's source form with n levels of
enclosing forms.
40
41   Stepping:
42   START Selects the CONTINUE restart if one exists and starts
43   single-stepping. Single stepping affects only code
compiled with
44   under high DEBUG optimization quality. See User
Manual for details.
45   STEP  Steps into the current form.
46   NEXT  Steps over the current form.
47   OUT   Stops stepping temporarily, but resumes it when the
topmost frame that
48   was stepped into returns.
49   STOP  Stops single-stepping.
50
51   ...
52
53   0] list-locals
54   SB-DEBUG::ARG-0 = "cat"

```

```

55 SB-DEBUG::ARG-1    =    1
56
57 0] backtrace 2
58
59 Backtrace for: #<SB-THREAD:THREAD "main thread" RUNNING
{1002AC32F3}>
60 0: (SB-KERNEL:TWO-ARG-+ "cat" 1)
61 1: (MY-ADD-ONE "cat")
62 0] :0
63
64 *
```

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

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.

Loading your Working Environment Quickly

When you start using Common Lisp for large projects, you will likely have

many files to load into your Lisp environment when you start working. Most Common Lisp implementations have a function 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.

Another good technique is to create a Lisp image containing all the code and data for all your projects. There is an example of this in the first section of the [Chapter on NLP](#). In this example, it takes a few minutes to load the code and data for my NLP (natural language processing) library so when I am working with it I like to be able to quickly load a SBCL Lisp image.

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

Functional Programming Concepts

There are two main styles for doing Common Lisp development. Object oriented programming is well supported (see the [Chapter on CLOS](#)) as is functional programming. In a nut shell, functional programming means that we should write functions with no side effects. First let me give you a non-functional example with side effects:

```
1 (defun non-functional-example (car)
2   (set-color car "red"))
```

This example using CLOS is non-functional because we modify the value of an argument to the function. Some functional languages like the Lisp Clojure language and the Haskell language dissuade you from modifying arguments to functions. With Common Lisp you should make a decision on which approach you like to use.

Functional programming means that we avoid maintaining state inside of functions and treat data as immutable (i.e., once an object is created, it is never modified). We could modify the last example to be function by creating a new car object inside the function, copy the attributes of the car passed as an object, change the color to “red” of the new car object, and return the new car instance as the value of the function.

as the value of the function.

Functional programming prevents many types of programming errors, makes unit testing simpler, and makes programming for modern multi-core CPUs easier because read-only objects are inherently thread safe. Modern best practices for the Java language also prefer immutable data objects and a functional approach.

Defining Lisp Functions

In the previous 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:

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

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

In Chapter 1, we saw an example of using 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:

```
1 * (defun test (x &aux y)  
2     (setq y (list x x))  
3     y)
```

```

4 TEST
5 * (test 'cat)
6 (CAT CAT)
7 * (test 3.14159)
8 (3.14159 3.14159)

```

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

```

1 * (defun test (x)
2     (let ((y (list x x)))
3         y))
4 TEST
5 * (test "the dog bit the cat")
6 ("the dog bit the cat" "the dog bit the cat")
7 *

```

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.

```

1 * (defun test (a &optional b (c 123))
2     (format t "a=~A b=~A c=~A~%" a b c))
3 TEST
4 * (test 1)
5 a=1 b=NIL c=123
6 NIL
7 * (test 1 2)
8 a=1 b=2 c=123
9 NIL
10 * (test 1 2 3)
11 a=1 b=2 c=3
12 NIL
13 * (test 1 2 "Italian Greyhound")
14 a=1 b=2 c=Italian Greyhound
15 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:

```

1 * (defun test (a &key b c)
2     (format t "a=~A b=~A c=~A~%" a b c))
3 TEST
4 * (test 1)
5 a=1 b=NIL c=NIL
6 NIL
7 * (test 1 :c 3.14159)
8 a=1 b=NIL c=3.14159
9 NIL
10 * (test "cat" :b "dog")
11 a=cat b=dog c=NIL
12 NIL
13 *
```

Using Lambda Forms

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:

```

1 * (funcall 'print 'cat)
2 CAT
3 CAT
4 * (funcall '+ 1 2)
5 3
6 * (funcall #'- 2 3)
7 -1
```


8 *

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**:

```
1 (defun test ()
2   (let ((my-func
3         (lambda (x) (+ x 1))))
4     (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:

```
1 * (test)
2 2
3
4 *
```

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

```
1 * (defvar f1 #'(lambda (x) (+ x 1)))
2
3 F1
4 * (funcall f1 100)
5
6 101
7 * (funcall #'print 100)
8
9 100
10 100
```

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).

.. . — .

Using Recursion

Later, 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:

```
1 ;; a simple loop using recursion
2
3 (defun recursion1 (value)
4   (format t "entering recursion1(~A)~%" value)
5   (if (< value 5)
6       (recursion1 (1+ value))))
```

This example is simple, 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:

```
1 * (load "recursion1.lisp")
2 ;; Loading file recursion1.lisp ...
3 ;; Loading of file recursion1.lisp is finished.
4 T
5 * (recursion1 0)
6 entering recursion1(0)
7 entering recursion1(1)
8 entering recursion1(2)
9 entering recursion1(3)
10 entering recursion1(4)
11 entering recursion1(5)
12 NIL
13 * (recursion1 -3)
14 entering recursion1(-3)
15 entering recursion1(-2)
16 entering recursion1(-1)
```

```
17 entering recursion1(0)
18 entering recursion1(1)
19 entering recursion1(2)
20 entering recursion1(3)
21 entering recursion1(4)
22 entering recursion1(5)
23 NIL
24 * (recursion1 20)
25 entering recursion1(20)
26 NIL
27 *
```

Why did the call on line 24 not loop via recursion? Because the input argument is not less than 5, no recursion occurs.

Closures

We have seen that functions can 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:

```
1 (let* ((fortunes
2         '("You will become a great Lisp Programmer"
3           "The force will not be with you"
4           "Take time for meditation")))
5       (len (length fortunes))
6       (index 0))
7   (defun fortune ()
8     (let ((new-fortune (nth index fortunes)))
9       (setq index (1+ index))
10      (if (>= index len) (setq index 0))
11      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:

```
1 * (load "closure1.lisp")
2 ;; Loading file closure1.lisp ...
3 ;; Loading of file closure1.lisp is finished.
4 T
5 * (fortune)
6 "You will become a great Lisp Programmer"
7 * (fortune)
8 "The force will not be with you"
9 * (fortune)
10 "Take time for meditation"
11 * (fortune)
12 "You will become a great Lisp Programmer"
13 *
```

Quicklisp

For several decades managing packages and libraries was a manual process when developing Lisp systems. I used to package the source code for specific versions of libraries as part of my Common Lisp projects. Early package management systems `mk-defsystem` and `ASDF` were very useful, but I did not totally give up my practice keeping third party library source code with my projects until Zach Beane created the [Quicklisp package system](#). You will need to have Quicklisp installed for many of the examples later in this book so please take the time to install it right now as per the instructions on the Quicklisp web site.

Using Quicklisp to Find Packages

We will need the Common Lisp `USOCKET` library later in the [Chapter on Network Programming](#) so we will install it now using Quicklisp. Please use the following example:

```
1 * (ql:quickload :hunchentoot)
2 To load "hunchentoot":
3   Load 1 ASDF system:
4     hunchentoot
5 ; Loading "hunchentoot"
6 .....
7 (:HUNCHENTOOT)
```

The first time you **ql:quickload** a library you may see additional printout. In most of the rest of this book, when I install or use a package by calling the **ql:quickload** function I do not show the output from this function when in the repl listings.

Now, we can use the fantastically useful Common Lisp function **apropos** to see what was just installed:

```
1 * (apropos "hunchentoot")
2
```

```

3 HUNCHENTOOT : : *CLOSE-HUNCHENTOOT-STREAM* (bound)
4 HUNCHENTOOT : : *HUNCHENTOOT-DEFAULT-EXTERNAL-FORMAT* (bound)
5 HUNCHENTOOT : : *HUNCHENTOOT-STREAM*
6 HUNCHENTOOT : : *HUNCHENTOOT-VERSION* (bound)
7 HUNCHENTOOT : HUNCHENTOOT-CONDITION
8 HUNCHENTOOT : HUNCHENTOOT-ERROR (fbound)
9 HUNCHENTOOT : : HUNCHENTOOT-OPERATION-NOT-IMPLEMENTED-OPERATION
(fbound)
10 HUNCHENTOOT : : HUNCHENTOOT-SIMPLE-ERROR
11 HUNCHENTOOT : : HUNCHENTOOT-SIMPLE-WARNING
12 HUNCHENTOOT : : HUNCHENTOOT-WARN (fbound)
13 HUNCHENTOOT : HUNCHENTOOT-WARNING
14 HUNCHENTOOT-ASD : *HUNCHENTOOT-VERSION* (bound)
15 HUNCHENTOOT-ASD : : HUNCHENTOOT
16 : HUNCHENTOOT (bound)
17 : HUNCHENTOOT-ASD (bound)
18 : HUNCHENTOOT-DEV (bound)
19 : HUNCHENTOOT-NO-SSL (bound)
20 : HUNCHENTOOT-TEST (bound)
21 : HUNCHENTOOT-VERSION (bound)
22 *

```

As long as you are thinking about the new tool Quicklisp that is now in your tool chest, you should install the rest of the packages and libraries that you will need for working through the rest of this book. I will show the statements needed to load more libraries without showing the output printed in the repl as each package is loaded:

```

1 (ql:quickload "clsql")
2 (ql:quickload "clsql-postgresql")
3 (ql:quickload "clsql-mysql")
4 (ql:quickload "clsql-sqlite3")
5 (ql:quickload :drakma)
6 (ql:quickload :hunchentoot)
7 (ql:quickload :cl-json)
8 (ql:quickload "clouchdb") ;; for CouchDB access

```

You need to have the Postgres and MySQL client developer libraries installed on

your system for the **clsql-postgresql** and **clsql-mysql** installations to work.

Using Quicklisp to Configure Emacs and Slime

I assume that you have Emacs installed on your system. In a repl you can setup the Slime package that allows Emacs to connect to a running Lisp environment:

```
1 (ql:quickload "quicklisp-slime-helper")
```

Pay attention to the output in the repl. On my system the output contained the following:

```
1 [package quicklisp-slime-helper]
2 slime-helper.el installed in "Usersmarkw/quicklisp/slime-
  helper.el"
3
4 To use, add this to your ~/.emacs:
5
6 (load (expand-file-name "~/quicklisp/slime-helper.el"))
7 ;; Replace "sbcl" with the path to your implementation
8 (setq inferior-lisp-program "sbcl")
```

If you installed **rlwrap** and defined an alias for running SBCL, make sure you set the inferior lisp program to the absolute path of the SBCL executable; on my system I set the following in my **.emacs** file:

```
1 (setq inferior-lisp-program "Usersmarkw/sbcl/sbcl")
```

I am not going to cover using Emacs and Slime: there are many good tutorials on the web you can read.

Defining Common Lisp Macros

We saw in the last chapter 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.

Example Macro

The file **src/macro1.lisp** contains both a simple macro and a function that uses the macro. This macro example is a bit contrived since it could be just a function definition, but it does show the process of creating and using a macro. We are using the **gensym** function to define a new unique symbol to reference a temporary variable:

```
1 ;; first simple macro example:
2
3 (defmacro double-list (a-list)
4   (let ((ret (gensym)))
5     `(let ((,ret nil))
6       (dolist (x ,a-list)
7         (setq ,ret (append ,ret (list x x))))
8       ,ret)))
9
10 ;; use the macro:
11
12 (defun test (x)
13   (double-list x))
```

The backquote character seen at the beginning of line 5 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 the next chapter but for now it is sufficient to understand that **dolist** is used to iterate through the top-level elements of a list, for example:

```
1 * (dolist (x '("the" "cat" "bit" "the" "rat"))
2     (print x))
3 "the"
4 "cat"
5 "bit"
6 "the"
7 "rat"
8 NIL
9 *
```

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

```
1 * (load "macro1.lisp")
2 ;; Loading file macro1.lisp ...
3 ;; Loading of file macro1.lisp is finished.
4 T
5 * (test '(1 2 3))
6 (1 1 2 2 3 3)
7 *
```

Using the Splicing Operator

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

```
1 ;; another macro example that uses ,@:
2
3 (defmacro double-args (&rest args)
4   `(let ((ret nil))
5     (dolist (x ,@args)
6       (setq ret (append ret (list x x))))
7     ret))
8
```

```

9 ;; use the macro:
10
11 (defun test (&rest x)
12   (double-args x))

```

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

Using macroexpand-1

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

```

1 * (defmacro double (a-number)
2     (list '+ a-number a-number))
3 DOUBLE
4 * (macroexpand-1 '(double n))
5 (+ N N) ;
6 T
7 *

```

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.”

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 that I almost never use in my own code because it does not look “Lisp like.” I don’t use the loop macro in this book. Many programmers do like the loop macro so you are likely to see it when reading other people’s code.

dolist

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

```
1      (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:

```
1 * (dolist (a '(1 2) 'done) (print a))
2 1
3 2
4 DONE
5 * (dolist (a '(1 2)) (print a))
6 1
7 2
8 NIL
9 *
```

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.

dotimes

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

```
1      (dotimes (an-index-variable max-index-plus-one [optional-
result-value])
2          ...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:

```
1 * (dotimes (i 3 "all-done-with-test-dotimes-loop") (print i))
2
3 0
4 1
5 2
6 "all-done-with-test-dotimes-loop"
7 *
```

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

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:

```
1  (do ((variable-1 variable-1-init-value variable-1-update-
expression)
2      (variable-2 variable-2-init-value variable-2-update-
expression)
3      .
```

```

4          .
5          (variable-N variable-N-init-value variable-N-update-
expression))
6          (loop-termination-test loop-return-value)
7          optional-variable-declarations
8          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

```

1 (do ((i 0 (1+ i)))
2      ((> i 3) "value-of-do-loop")
3      (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**:

```

1 * (load "do1.lisp")
2 ;; Loading file do1.lisp ...
3 0
4 1
5 2
6 3
7 ;; Loading of file do1.lisp is finished.
8 T
9 *

```

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.

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*:

```
1 > *package*  
2 #<PACKAGE COMMON-LISP-USER>  
3 >
```

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.

```
1 * (defun foo1 () "foo1")  
2  
3 F001  
4 (defpackage "MY-NEW-PACKAGE"  
5   (:use :cl)  
6   (:nicknames "P1")  
7   (:export "F002"))  
8  
9 #<PACKAGE "MY-NEW-PACKAGE">  
10 * (in-package my-new-package)  
11  
12 #<PACKAGE "MY-NEW-PACKAGE">  
13 * (foo1)  
14 ; in: F001  
15 ; (MY-NEW-PACKAGE::F001)  
16 ;  
17 ; caught STYLE-WARNING:  
18 ; undefined function: F001
```

```
19 ;
20 ; compilation unit finished
21 ;   Undefined function:
22 ;       F001
23 ;   caught 1 STYLE-WARNING condition
24
25 debugger invoked on a UNDEFINED-FUNCTION:
26   The function MY-NEW-PACKAGE::F001 is undefined.
27
28 Type HELP for debugger help, or (SB-EXT:EXIT) to exit from
SBCL.
29
30 restarts (invokable by number or by possibly-abbreviated name):
31   0: [ABORT] Exit debugger, returning to top level.
32
33 ("undefined function")
34 0] :a
35
36 * (common-lisp-user::foo1)
37
38 "foo1"
39 * (in-package :common-lisp-user)
40
41 #<PACKAGE "COMMON-LISP-USER">
42 * (foo2)
43
44 ; in: F002
45 ;   (F002)
46
47 debugger invoked on a UNDEFINED-FUNCTION:
48   The function COMMON-LISP-USER::F002 is undefined.
49
50 Type HELP for debugger help, or (SB-EXT:EXIT) to exit from
SBCL.
51
52 restarts (invokable by number or by possibly-abbreviated name):
53   0: [ABORT] Exit debugger, returning to top level.
54
55 ("undefined function")
```



```
56 0] a
57
58 * (my-new-package::foo2)
59
60 "foo2"
61 * (p1::foo2)
62
63 "foo2"
```

Since we specified a nickname in the **defpackage** expression, Common Lisp uses the nickname (in this case **P1** at the beginning of the expression prompt when we switch to 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 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.

When I use a package I usually 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, 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.

Input and Output

We will see that 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, 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 default streams are connected to the Lisp listener that we discussed in Chapter 2.

The Lisp read and readline 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 1 2 3
2 4 "the cat bit the rat"
3      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 can 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 function **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:

```
1 (defun read-test-1 ()
2   "read a maximum of 1000 expressions from the file 'test.dat'"
3   (with-open-file
4     (input-stream "test.dat" :direction :input)
5     (dotimes (i 1000)
6       (let ((x (read input-stream nil nil)))
7         (if (null x) (return)) ;; break out of the 'dotimes'
loop
8         (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):

```
1 * (load "read-test-1.lisp")
2 ;; Loading file read-test-1.lisp ...
3 ;; Loading of file read-test-1.lisp is finished.
4 T
5 * (read-test-1)
6 next expression in file: 1
7 next expression in file: 2
8 next expression in file: 3
9 next expression in file: 4
10 next expression in file: "the cat bit the rat"
11 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 generally portable across all operating systems. Instead, we could have created a pathname object and passed that instead. The **pathname**

function can take eight 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:

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

Here, we are specifying that we should look 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 all Common Lisp implementations.

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

```
1  (defun readline-test ()
2    "read a maximum of 1000 expressions from the file 'test.dat'"
3    (with-open-file
4      (input-stream "test.dat" :direction :input)
5      (dotimes (i 1000)
6        (let ((x (read-line input-stream nil nil)))
7          (if (null x) (return)) ;; break out of the 'dotimes'
loop
8          (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:

```
1  * (load "readline-test.lisp")
2  ;; Loading file readline-test.lisp ...
3  ;; Loading of file readline-test.lisp is finished.
4  T
5  * (readline-test)
```

```

6 next line in file: "1 2 3"
7 next line in file: "4 \"the cat bit the rat\""
8 NIL
9 *

```

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):

```

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

```

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

```

1 * (load "read-from-string-test.lisp")
2 ;; Loading file read-from-string-test.lisp ...
3 ;; Loading of file read-from-string-test.lisp is finished.
4 T
5 * (read-from-string-test)
6 next expression in string: 1
7 next expression in string: 2
8 next expression in string: "My parrot is named Brady."
9 next expression in string: (11 22)
10 NIL
11 *

```

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.

Lisp Printing Functions

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:

```
1 * (print "testing")
2
3 "testing"
4 "testing"
5 * (print "testing" *standard-output*)
6
7 "testing"
8 "testing"
9 *
```

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:

```
1 * (print "testing")
2
3 "testing"
4 "testing"
5 * (princ "testing")
6 testing
7 "testing"
8 *
```

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:

```
1 * (let ((l1 '(1 2))
2         (x 3.14159))
3     (format nil "AA" l1 x))
4 "(1 2)3.14159"
5 *
```

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:

```
1 (with-open-file (out-stream "test1.dat"
2                           :direction :output
3                           :if-exists :supersede)
4     (print "the cat ran down the road" out-stream)
5     (format out-stream "1 + 2 is: A%" (+ 1 2))
6     (princ "Stoking!!" out-stream)
7     (terpri out-stream))
```

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

```
1 % cat test1.dat
2
3 "the cat ran down the road" 1 + 2 is: 3
4 Stoking!!
```

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

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 used this CLOS class about ten years ago in a demo for my commercial natural language processing product to automatically generate demo web pages.

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 see how to use this example CLOS 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 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.

Example of Using a CLOS Class

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

```
1 (defun test (&aux x)
2     (setq x (make-instance 'HTMLstream))
3     (set-header x "test page")
4     (add-element x "test text - this could be any element")
5     (add-table
6         x
7         '(("<b>Key phrase</b>" "<b>Ranking value</b>")
8           ("this is a test" 3.3)))
```



```
9      (get-htmlstring x))
```

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

```
1      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
- 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.

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:

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

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

```
1 (defclass HTMLstream ()  
2   ((out :accessor out))  
3   (: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 of local variable to one or more class slot accessors. We will list the entire method then discuss it:

```
1 (defmethod set-header ((ho HTMLstream) title)  
2   (with-accessors  
3     ((out out))  
4     ho  
5     (setf out (make-string-output-stream))  
6     (princ "<HTML><head><title>" out)  
7     (princ title out)  
8     (princ "</title></head><BODY>" out)  
9     (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 look at just 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 on Input and Output to write HTML text to the string output stream:

```
1 (defmethod add-H1 ((ho HTMLstream) some-text)
2   (with-accessors
3     ((out out))
4     ho
5     (princ "<H1>" out)
6     (princ some-text out)
7     (princ "</H1>" out)
8     (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:

```
1 (defmethod add-element ((ho HTMLstream) element)
2   (with-accessors
3     ((out out))
4     ho
```

```

5      (princ element out)
6      (terpri out)))

```

The method **add-table** converts 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:

```

1  (defmethod add-table ((ho HTMLstream) table-data)
2    (with-accessors
3      ((out out))
4      ho
5      (princ "<TABLE BORDER=\"1\" WIDTH=\"100%\">" out)
6      (dolist (d table-data)
7        (terpri out)
8        (princ " <TR>" out)
9        (terpri out)
10       (dolist (w d)
11         (princ " <TD>" out)
12         (let ((str (princ-to-string w)))
13           (setq str (string-left-trim '(#\() str))
14           (setq str (string-right-trim '(#\)) str))
15           (princ str out))
16         (princ "</TD>" out)
17         (terpri out))
18       (princ " </TR>" out)
19       (terpri out))
20      (princ "</TABLE>" out)
21      (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**:

```
1 (defmethod get-htmlstring ((ho HTMLstream))
2   (with-accessors
3     ((out out))
4     ho
5     (princ "</BODY></HTML>" out)
6     (terpri out)
7     (get-output-stream-string out)))
```

CLOS is a rich framework for object oriented programming, providing a superset of features found in languages like Java, Ruby, and Smalltalk. I have barely scratched the surface in this short CLOS example for generating HTML. Later in the book, whenever you see calls to **make-instance** that lets you know we are using CLOS even if I don't specifically mention CLOS in the examples.

Network Programming

Distributed computing is pervasive: you need to look no further than 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! The previous editions of this book provided low level socket network programming examples. I decided that for this new edition, I would remove those examples and instead encourage you to “move further up the food chain” and work at a higher level of abstraction that makes sense for the projects you will likely be developing. Starting in the 1980s, a lot of my work entailed low level socket programming for distributed networked applications. As I write this, it is 2013, and there are better ways to structure distributed applications.

Specifically, since many of the examples later in this book fetch information from the web and linked data sources, we will start by learning how to use Edi Weitz’s [Drakma HTTP client library](#). In order to have a complete client server example we will also look briefly at Edi Weitz’s [Hunchentoot web server](#) that uses JSON as a data serialization format. I used to use XML for data serialization but JSON has many advantages: easier for a human to read and it plays nicely with Javascript code and some data stores like Postgres (new in versions 9.x), MongoDB, and CouchDB that support JSON as a native data format.

The code snippets in the first two sections of this chapter are derived from examples in the Drackma and Hunchentoot documentation.

An introduction to Drakma

Edi Weitz’s [Drakma library](#) supports fetching data via HTTP requests. As you can see in the Drakma documentation, you can use this library for authenticated HTTP requests (i.e., allow you to access web sites that require a login), support HTTP GET and PUT operations, and deal with cookies. The top level API that we will use is **drakma:http-request** that returns multiple values. In the following example, I want only the first three values, and ignore the others like the original URI that was fetched and an IO stream object. We use the built-in Common Lisp macro **multiple-value-setq**:

```

1 * (ql:quickload :drakma)
2 * (multiple-value-setq
3     (data http-response-code headers)
4     (drakma:http-request "http://markwatson.com"))

```

I manually formatted the last statement I entered in the last repl listing and I will continue to manually edit the repl listings in the rest of this book to make them more easily readable.

The following shows some of the data bound to the variables **data**, **http-response-code**, and **headers**:

```

1 * data
2
3 "<!DOCTYPE html>"
4 <html>
5   <head>
6     <title>Mark Watson: Consultant and Author</title>

```

The value of **http-response-code** is 200 which means that there were no errors:

```

1 * http-response-code
2
3 200

```

The HTTP response headers will be useful in many applications; for fetching the home page of my web site the headers are:

```

1 * headers
2
3 ( (:SERVER . "nginx/1.1.19")
4   (:DATE . "Fri, 05 Jul 2013 15:18:27 GMT")
5   (:CONTENT-TYPE . "text/html; charset=utf-8")
6   (:TRANSFER-ENCODING . "chunked")
7   (:CONNECTION . "close")
8   (:SET-COOKIE
9     .
10    "ring-session=cec5d7ba-e4da-4bf4-b05e-aff670e0dd10;Path=/") )

```

An introduction to Hunchentoot

Edi Weitz's [Hunchentoot project](#) is a flexible library for writing web applications and web services. We will also use Edi's CL-WHO library in this section for generating HTML from Lisp code. Hunchentoot will be installed the first time you quick load it in the example code for this section:

```
1 (ql:quickload "hunchentoot")
```

I will use only [easy handler framework](#) in the Hunchentoot examples in this section. I leave it to you to read the [documentation on using custom acceptors](#) after you experiment with the examples in this section.

The following code will work for both multi-threading installations of SBCL and single thread installations (e.g., some default installations of SBCL on OS X):

```
1 (ql:quickload :hunchentoot)
2 (ql:quickload :cl-who)
3
4 (in-package :cl-user)
5 (defpackage hdemo
6   (:use :cl
7         :cl-who
8         :hunchentoot))
9 (in-package :hdemo)
10
11 (defvar *h* (make-instance 'easy-acceptor :port 3000))
12
13 ;; define a handler with the arbitrary name my-greetings:
14
15 (define-easy-handler (my-greetings :uri "/hello") (name)
16   (setf (hunchentoot:content-type*) "text/html")
17   (with-html-output-to-string (*standard-output* nil :prologue
18     t)
19     (:html
20      (:head (:title "hunchentoot test"))
21      (:body
22       (:h1 "hunchentoot form demo")
23       (:form
```



```

23         :method :post
24         (:input :type :text
25             :name "name"
26             :value name)
27         (:input :type :submit :value "Submit your name"))
28         (:p "Hello " (str name))))))
29
30 (hunchentoot:start *h*)

```

In lines 5 through 9 we create an use a new package that includes support for generating HTML in Lisp code (CL-WHO) and the Hunchentoot library). On line 11 we create an instance of an easy acceptor on port 3000 that provides useful default behaviors for providing HTTP services.

The Hunchentoot macro **define-easy-handler** is used in lines 15 through 28 to define an HTTP request handler and add it to the easy acceptor instance. The first argument, **my-greetings** in this example, is an arbitrary name and the keyword **:uri** argument provides a URL pattern that the easy acceptor server object uses to route requests to this handler. For example, when you run this example on your computer, this URL routing pattern would handle requests like:

```

1 http://localhost:3000/hello

```

In lines 17 through 28 we are using the CL-WHO library to generate HTML for a web page. As you might guess, **:html** generates the outer `<html></html>` tags for a web page. Line 19 would generate HTML like:

```

1 <head>
2     <title>hunchentoot test</title>
3 </head>

```

Lines 22 through 27 generate an HTML input form and line 28 displays any value generated when the user entered text in the input field and clicked the submit button. Notice the definition of the argument **name** in line 1 in the definition of the easy handler. If the argument **name** is not defined, the **nil** value will be displayed in line 28 as an empty string.

You should run this example and access the generated web page in a web

browser, and enter text, submit, *etc.* You can also fetch the generated page HTML using the Drakma library that we saw in the last section. Here is a code snippet using the Drakma client library to access this last example:

```
1 * (drakma:http-request "http://127.0.0.1:3000/hello?name=Mark")
2
3 "Hello Mark"
4 200
5 (:CONTENT-LENGTH . "10")
6 (:DATE . "Fri, 05 Jul 2013 15:57:22 GMT")
7 (:SERVER . "Hunchentoot 1.2.18")
8 (:CONNECTION . "Close")
9 (:CONTENT-TYPE . "text/plain; charset=utf-8"))
10 #<PURI:URI http://127.0.0.1:3000/hello?name=Mark>
11 #<FLEXI-STREAMS:FLEXI-IO-STREAM {10095654A3}>
12 T
13 "OK"
```

We will use Drakma later in this book for several examples.

Complete REST Client Server Example Using JSON for Data Serialization

A reasonable way to build modern distributed systems is to write REST web services that serve JSON data to client applications. These client applications might be rich web apps written in Javascript, other web services, and applications running on smartphones that fetch and save data to a remote web service.

We will use the **cl-json** Quicklisp package to encode Lisp data into a string representing JSON encoded data. Here is a quick example:

```
1 * (ql:quickload :cl-json)
2 * (defvar y (list (list '(cat . "the cat ran") '(dog . 101)) 1
2 3 4 5))
3
4 Y
5 * y
6
```

```
7 (((CAT . "the cat ran") (DOG . 101)) 1 2 3 4 5)
8 * (json:encode-json-to-string y)
9 "[{"cat": "the cat ran", "dog": 101}, 1, 2, 3, 4, 5]"
```

The following list shows the contents of the file **src/web-hunchentoot-json.lisp**:

```
1 (ql:quickload :hunchentoot)
2 (ql:quickload :cl-json)
3
4 (defvar *h* (make-instance 'hunchentoot:easy-acceptor :port
5 3000))
6 ;; define a handler with the arbitrary name my-greetings:
7
8 (hunchentoot:define-easy-handler (animal :uri "/animal") (name)
9   (print name)
10   (setf (hunchentoot:content-type*) "text/plain")
11   (cond
12     ((string-equal name "cat")
13      (json:encode-json-to-string
14        (list
15          (list
16            '(average_weight . 10)
17            '(friendly . nil))
18          "A cat can live indoors or outdoors.))))
19     ((string-equal name "dog")
20      (json:encode-json-to-string
21        (list
22          (list
23            '(average_weight . 40)
24            '(friendly . t))
25          "A dog is a loyal creature, much valued by humans.))))
26     (t
27      (json:encode-json-to-string
28        (list
29          ()
30          "unknown type of animal")))))
31
```

```
32 (hunchentoot:start *h*)
```

This example is very similar to the web application example in the last section. The difference is that this application is not intended to be viewed on a web page because it returns JSON data as HTTP responses. The easy handler definition on line 8 specifies a handler argument **name**. In lines 12 and 19 we check to see if the value of the argument **name** is “cat” or “dog” and if it is, we return the appropriate JSON example data for those animals. If there is no match, the default **cond** clause starting on line 26 returns a warning string as a JSON encoded string.

While running this test service, in one repl, you can use the Drakma library in another repl to test it (not all output is shown in the next listing):

```
1 * (ql:quickload :drakma)
2 * (drakma:http-request "http://127.0.0.1:3000/animal?name=dog")
3
4 "[{"average_weight":40,
5   "friendly":true},
6   "A dog is a loyal creature, much valued by humans."]"
7 200
8 * (drakma:http-request "http://127.0.0.1:3000/animal?name=cat")
9
10 "[{"average_weight":10,
11   "friendly":null},
12   "A cat can live indoors or outdoors."]"
13 200
```

You can use the **cl-json** library to decode a string containing JSON data to Lisp data:

```
1 * (ql:quickload :cl-json)
2 To load "cl-json":
3   Load 1 ASDF system:
4     cl-json
5 ; Loading "cl-json"
6 .
7 (:CL-JSON)
```

```

8 * (cl-json:decode-json-from-string
9     (drakma:http-request "http://127.0.0.1:3000/animal?
name=dog"))
10
11 (((:AVERAGE--WEIGHT . 40) (:FRIENDLY . T))
12  "A dog is a loyal creature, much valued by humans.")

```

For most of my work, REST web services are “read-only” in the sense that clients don’t modify state on the server. However, there are use cases where a client application might want to; for example, letting clients add new animals to the last example.

```

1 (defparameter *animal-hash* (make-hash-table))
2
3 ;; handle HTTP POST requests:
4 (hunchentoot:define-easy-handler (some-handler :uri "/add")
(json-data)
5   (setf (hunchentoot:content-type*) "text/plain")
6   (let* ((data-string (hunchentoot:raw-post-data :force-text
t))
7           (data (cl-json:decode-json-from-string json-data))
8           ;; assume that the name of the animal is a hashed
value:
9           (animal-name (gethash "name" data)))
10    (setf (gethash animal-name *animal-hash*) data))
11    "OK")

```

In line 4 we are defining an additional easy handler with a handler argument **json-data**. This data is assumed to be a string encoding of JSON data which is decoded into Lisp data in lines 6 and 7. We save the data to the global variable *animal-hash*.

In this example, we are storing data sent from a client in an in-memory hash table. In a real application new data might be stored in a database.

Accessing Relational Databases

There are good options for accessing relational databases from Common Lisp. Personally I almost always use Postgres and in the past I used either native foreign client libraries or the socket interface to Postgres. Recently, I decided to switch to [CLSQL](#) which provides a common interface for accessing Postgres, MySQL, SQLite, and Oracle databases. There are also several recent forks of CLSQL on github. We will use CLSQL in examples in this book. Hopefully while reading the [Chapter on Quicklisp](#) you installed CLSQL and the back end for one or more databases that you use for your projects. If you have not installed CLSQL yet, then please install it now:

```
1 (ql:quickload "clsql")
```

You also need to install one or more CLSQL backends, depending on which relational databases you use:

```
1 (ql:quickload "clsql-postgresql")
2 (ql:quickload "clsql-mysql")
3 (ql:quickload "clsql-sqlite3")
```

While I often prefer hand crafting SQL queries, there seems to be a general movement in software development towards the data mapper or active record design patterns. CLSQL provides Object Relational Mapping (ORM) functionality to CLOS.

You will need to create a new database **news** in order to follow along with the examples in this chapter and later in this book. I will use Postgres for examples in this chapter and use the following to create a new database (my account is “markw” and the following assumes that I have Postgres configured to not require a password for this account when accessing the database from “localhost”):

```
1 -> ~ psql
2 psql (9.1.4)
```

```
3 Type "help" for help.
4 markw=# create database news;
5 CREATE DATABASE
```

We will use three example programs that you can find in the **src** directory in the book repository on github:

- `clsql_create_news_schema.lisp` to create table “articles” in database “news”
- `clsql_write_to_news.lisp` to write test data to table “articles”
- `clsql_read_from_news.lisp`

The following listing shows the file **src/clsql_create_news_schema.lisp**:

```
1 (ql:quickload :clsql)
2 (ql:quickload :clsql-postgresql)
3
4 ;; Postgres connection specification:
5 ;;      (host db user password &optional port options tty).
6 ;; The first argument to clsql:connect is a connection
7 ;; specification list:
8
9 (clsql:connect '("localhost" "news" "markw" nil)
10               :database-type :postgresql)
11
12 (clsql:def-view-class articles ()
13   ((id
14     :db-kind :key
15     :db-constraints :not-null
16     :type integer
17     :initarg :id)
18    (uri
19     :accessor uri
20     :type (string 60)
21     :initarg :uri)
22    (title
23     :accessor title
24     :type (string 90)
25     :initarg :title)
26    (text
```

```

27      :accessor text
28      :type (string 500)
29      :nulls-ok t
30      :initarg :text)))
31
32 (defun create-articles-table ()
33   (clsql:create-view-from-class 'articles))

```

In this repl listing, we create the database table “articles” using the function **create-articles-table** that we just defined:

```

1 ->  src git:(master) sbcl
2 (running SBCL from: /Users/markw/sbcl)
3 * (load "clsql_create_news_schema.lisp")
4 * (create-articles-table)
5 NOTICE: CREATE TABLE / PRIMARY KEY will create implicit index
6         "article_pk" for table "articles"
7 T
8 *

```

The following listing shows the file **src/clsql_write_to_news.lisp**:

```

1 (ql:quickload :clsql)
2 (ql:quickload :clsql-postgresql)
3
4 ;; Open connection to database and create CLOS class and
database view
5 ;; for table 'articles':
6 (load "clsql_create_news_schema.lisp")
7
8 (defvar *a1*
9   (make-instance
10    'article
11    :uri "http://test.com"
12    :title "Trout Season is Open on Oak Creek"
13    :text "State Fish and Game announced the opening of trout
season"))
14

```



```

15 (clsq:update-records-from-instance *a1*)
16 ;; modify a slot value and update database:
17 (setf (slot-value *a1* 'title) "Trout season is open on Oak
Creek!!!")
18 (clsq:update-records-from-instance *a1*)
19 ;; warning: the last statement changes the "id" column in the
table

```

You should load the file **clsq_write_to_news.lisp** one time in a repl to create the test data. The following listing shows file **clsq_read_from_news.lisp**:

```

1 (ql:quickload :clsq)
2 (ql:quickload :clsq-postgresql)
3
4 ;; Open connection to database and create CLOS class and
database view
5 ;; for table 'articles':
6 (load "clsq_create_news_schema.lisp")
7
8 (defun pp-article (article)
9   (format t
10     "~%URI: S %Title: S %Text: S %"
11     (slot-value article 'uri)
12     (slot-value article 'title)
13     (slot-value article 'text)))
14
15 (dolist (a (clsq:select 'article))
16   (pp-article (car a)))

```

Loading the file **clsq_read_from_news.lisp** produces the following output:

```

1 URI: "http://test.com"
2 Title: "Trout season is open on Oak Creek!!!"
3 Text: "State Fish and Game announced the opening of trout season"
4
5 URI: "http://example.com"
6 Title: "Longest day of year"
7 Text: "The summer solstice is on Friday."

```

You can also embed SQL where clauses in queries:

```
1 (dolist (a (clsql:select 'article :where "title like
  '%season%'"'))
2   (pp-article (car a)))
```

which produces this output:

```
1 URI: "http://test.com"
2 Title: "Trout season is open on Oak Creek!!!"
3 Text: "State Fish and Game announced the opening of
4       trout season"
```

In this example, I am using a SQL **like** expression to perform partial text matching.

Using MongoDB, CouchDB, and Solr NoSQL Data Stores

Non-relational data stores are commonly used for applications that don't need either full relational algebra or must scale.

Brewer's CAP theorem states that a distributed data storage system comprised of multiple nodes can be robust to two of three of the following guarantees: all nodes always have a **C**onsistent view of the state of data, general **A**vailability of data if not all nodes are functioning, and **P**artition tolerance so clients can still communicate with the data storage system when parts of the system are unavailable because of network failures. The basic idea is that different applications have different requirements and sometimes it makes sense to reduce system cost or improve scalability by easing back on one of these requirements.

A good example is that some applications may not need transactions (the first guarantee) because it is not important if clients sometimes get data that is a few seconds out of date.

MongoDB allows you to choose consistency vs. availability vs. efficiency.

I cover the Solr indexing and search service (based on Lucene) both because a Solr indexed document store is a type of NoSQL data store and also because I believe that you will find Solr very useful for building systems, if you don't already use it.

We will design the examples in this chapter so that they can be reused in the last chapter in this book on [Information Gathering](#).

MongoDB

The following discussion of MongoDB is based on just my personal experience, so I am not covering all use cases. I have used MongoDB for:

- Small clusters of MongoDB nodes to analyze social media data, mostly text mining and sentiment analysis. In all cases for each application I ran

MongoDB with one write master (i.e., I wrote data to this one node but did not use it for reads) and multiple read-only slave nodes. Each slave node would run on the same server that was performing (usually) a single bit of analytics.

- Multiple very large independent clusters for web advertising. Problems faced included trying to have some level of consistency across data centers. Replica sets were used within each data center.
- Running a single node MongoDB instance for low volume data collection and analytics.

One of the advantages of MongoDB is that is very “developer friendly” because it supports ad-hoc document schemas and interactive queries. I mentioned that MongoDB allows you to choose consistency vs. availability vs. efficiency. When you perform MongoDB writes you can specify some granularity of what constitutes a “successful write” by requiring that a write is performed at a specific number of nodes before the client gets acknowledgement that the write was successful. This requirement adds overhead to each write operation and can cause writes to fail if some nodes are not available.

The [MongoDB online documentation](#) is very good. You don’t have to read it in order to have fun playing with the following Common Lisp and MongoDB examples, but if you find that MongoDB is a good fit for your needs after playing with these examples then you should read the documentation. I usually install MongoDB myself but it is sometimes convenient to use a hosting service. There are several well regarded services and I have used [MongoHQ](#).

At this time there is no official Common Lisp support for accessing MongoDB but there is a useful project by Alfons Haffmans’ [cl-mongo](#) that will allow us to write Common Lisp client applications and have access to most of the capabilities of MongoDB.

The file `src/mongo_news.lisp` contains the example code used in the next three sessions.

Adding Documents

The following repl listing shows the **cl-mongo** APIs for creating a new document, adding elements (attributes) to it, and inserting it in a MongoDB data store:

```

1 (ql:quickload "cl-mongo")
2
3 (cl-mongo:db.use "news")
4
5 (defun add-article (uri title text)
6   (let ((doc (cl-mongo:make-document)))
7     (cl-mongo:add-element "uri" uri doc)
8     (cl-mongo:add-element "title" title doc)
9     (cl-mongo:add-element "text" text doc)
10    (cl-mongo:db.insert "article" doc)))
11
12 ;; add a test document:
13 (add-article "http://test.com" "article title 1" "article text
14 1")

```

In this example, three string attributes were added to a new document before it was saved.

Fetching Documents by Attribute

We will start by fetching and pretty-printing all documents in the collection **articles** and fetching all articles a list of nested lists where the inner nested lists are document URI, title, and text:

```

1 (defun print-articles ()
2   (cl-mongo:pp (cl-mongo:iter (cl-mongo:db.find "article"
3 :all))))
4
5 ;; for each document, use the cl-mongo:get-element on
6 ;; each element we want to save:
7 (defun article-results->lisp-data (mdata)
8   (let ((ret '()))
9     ;;(print (list "size of result=" (length mdata)))
10    (dolist (a mdata)
11      ;;(print a)
12      (setf
13        ret
14        (cons
15          (list

```

```

15         (cl-mongo:get-element "uri" a)
16         (cl-mongo:get-element "title" a)
17         (cl-mongo:get-element "text" a))
18     ret)))
19 ret))
20
21 (defun get-articles ()
22   (article-results->lisp-data
23     (cadr (cl-mongo:db.find "article" :all))))

```

Output for these two functions looks like:

```

1 * (print-articles)
2
3 {
4   "_id" -> objectid(99778A792EBB4F76B82F75C6)
5   "uri"  -> http://test.com/3
6   "title" -> article title 3
7   "text"  -> article text 3
8 }
9
10 {
11   "_id" -> objectid(D47DEF3CFDB44DEA92FD9E56)
12   "uri"  -> http://test.com/2
13   "title" -> article title 2
14   "text"  -> article text 2
15 }
16
17 * (get-articles)
18
19 (("http://test.com/2" "article title 2" "article text 2")
20  ("http://test.com/3" "article title 3" "article text 3"))

```

Fetching Documents by Regular Expression Text Search

By reusing the function **article-results->lisp-data** defined in the last section, we can also search for JSON documents using regular expressions matching attribute values:

```

1 ;; find documents where substring 'str' is in the title:
2 (defun search-articles-title (str)
3   (article-results->lisp-data
4     (cadr
5       (cl-mongo:iter
6         (cl-mongo:db.find
7           "article"
8           (cl-mongo:kv
9             "title" // TITLE ATTRIBUTE
10            (cl-mongo:kv "$regex" str)) :limit 10))))))
11
12 ;; find documents where substring 'str' is in the text
13 element:
14 (defun search-articles-text (str)
15   (article-results->lisp-data
16     (cadr
17       (cl-mongo:db.find
18         "article"
19         (cl-mongo:kv
20           "text" // TEXT ATTRIBUTE
21           (cl-mongo:kv "$regex" str)) :limit 10))))))

```

I set the limit to return a maximum of ten documents. If you do not set the limit, this example code only returns one search result. The following repl listing shows the results from calling function **search-articles-text**:

```

1 * (SEARCH-ARTICLES-TEXT "text")
2
3 (("http://test.com/2" "article title 2" "article text 2")
4  ("http://test.com/3" "article title 3" "article text 3"))
5 * (SEARCH-ARTICLES-TEXT "3")
6
7 (("http://test.com/3" "article title 3" "article text 3"))

```

I find using MongoDB to be especially effective when experimenting with data and code. The schema free JSON document format, using interactive queries using the [mongo shell](#), and easy to use client libraries like **clouddb** for Common Lisp will let you experiment with a lot of ideas in a short period of

time. The following listing shows the use of the interactive **mongo shell**. The database **news** is the database used in the MongoDB examples in this chapter; you will notice that I also have other databases for other projects on my laptop:

```
1 -> src git:(master) mongo
2 MongoDB shell version: 2.4.5
3 connecting to: test
4 > show dbs
5 kbsportal          0.03125GB
6 knowledgespace    0.03125GB
7 local (empty)
8 mark_twitter      0.0625GB
9 myfocus           0.03125GB
10 news              0.03125GB
11 nyt               0.125GB
12 twitter           0.125GB
13 > use news
14 switched to db news
15 > show collections
16 article
17 system.indexes
18 > db.article.find()
19 { "uri" : "http://test.com/3",
20   "title" : "article title 3",
21   "text" : "article text 3",
22   "_id" : ObjectId("99778a792ebb4f76b82f75c6") }
23 { "uri" : "http://test.com/2",
24   "title" : "article title 2",
25   "text" : "article text 2",
26   "_id" : ObjectId("d47def3cfdb44dea92fd9e56") }
27 >
```

Line 1 of this listing shows starting the mongo shell. Line 4 shows how to list all databases in the data store. In line 13 I select the database “news” to use. Line 15 prints out the names of all collections in the current database “news”. Line 18 prints out all documents in the “articles” collection. You can read the [documentation for the mongo shell](#) for more options like selective queries, adding indices, *etc.*

When you run a MongoDB service on your laptop, also try the [admin interface on http://localhost:28017/](http://localhost:28017/).

CouchDB

CouchDB provides replication services so JSON documents you store with a CouchDB service on your laptop, for instance, can replicate to CouchDB services on your remote servers. For the rest of this chapter we will look at some code examples for using CouchDB from Common Lisp applications.

Building CouchDB from source takes a while since you need Erlang and Javascript dependencies. You can [download prebuilt binaries for Windows and Mac OS X](#) and on Linux use **sudo apt-get install couchdb**. I also have found it useful to use the CouchDB hosting service [Cloudant](#). Cloudant has open sourced a major piece of their infrastructure, a distributed version of CouchDB called BigCouch that you might want to consider if you need a very large distributed document data store.

Getting CouchDB Set Up

After you run a CouchDB service you can then use the web administration interface http://127.0.0.1:5984/_utils/index.html to create a new database **news** that we will use in these examples.

To install and use the [clouddb](#) CouchDB client library and use the new **news** database:

```
1 (ql:quickload "clouddb")
2 * clouddb:*couchdb*
3
4 #S(CLOUCHDB::DB
5   :HOST "localhost"
6   :PORT "5984"
7   :NAME "default"
8   :PROTOCOL "http"
9   :USER NIL
10  :PASSWORD NIL
11  :DOCUMENT-FETCH-FN NIL
12  :DOCUMENT-UPDATE-FN NIL)
13 * (setf clouddb:*couchdb*
```

```

14         (clouddb:make-db :name "news" :host "localhost"))
15
16 #S(CLOUCHDB::DB
17     :HOST "localhost"
18     :PORT "5984"
19     :NAME "news"
20     :PROTOCOL "http"
21     :USER NIL
22     :PASSWORD NIL
23     :DOCUMENT-FETCH-FN NIL
24     :DOCUMENT-UPDATE-FN NIL)

```

Notice that in line 7 that the database name is “default.” We want to change the database to “news” which we do in line 13. You can in a similar way change any other attributes in the **clouddb::clouddb** connection structure.

Adding Documents

CouchDB documents are stored as JSON data. The **clouddb** library lets us deal with documents as Lisp lists and converts to and from JSON for us.

We can create a document passing a list of lists to the function **clouddb:create-document**. The sub-lists are treated as hash table key/value pairs:

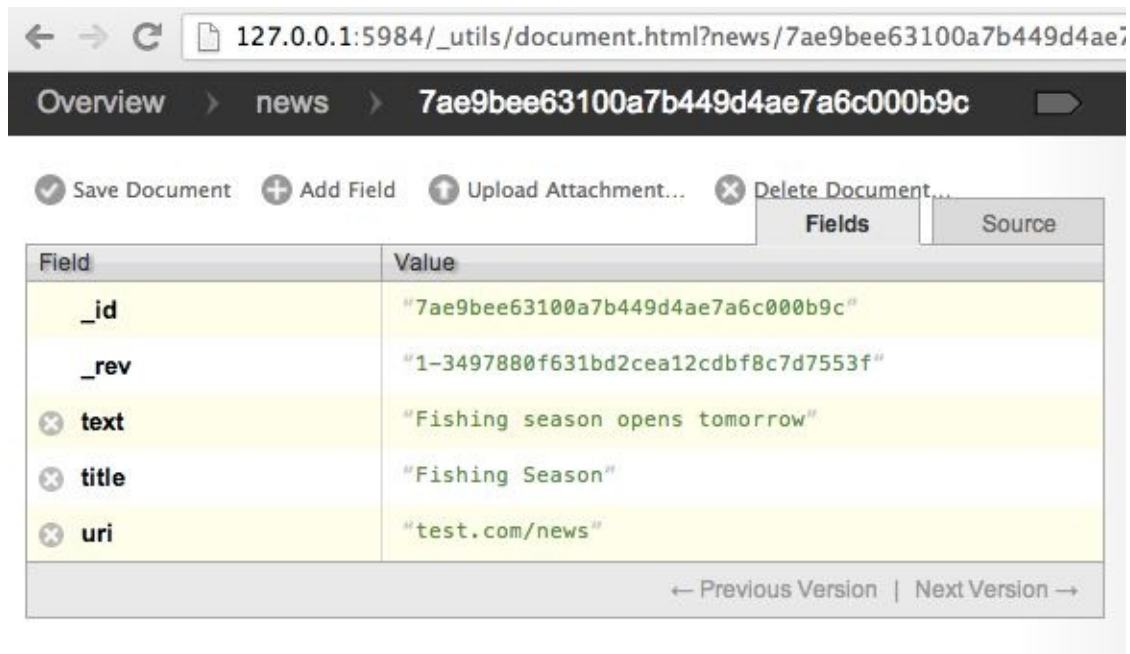
```

1 (defvar *d1*
2   (clouddb:create-document
3     '(("uri" . "test.com/news")
4       ("title" . "Fishing Season")
5       ("text" . "Fishing season opens tomorrow"))))
6 * *d1*
7
8 ( (:|ok| . T) (:|id| . "7ae9bee63100a7b449d4ae7a6c000b9c")
9   (:|rev| . "1-3497880f631bd2cea12cdbf8c7d7553f"))
10 * (clouddb:get-document "7ae9bee63100a7b449d4ae7a6c000b9c")
11
12 ( (:|_id| . "7ae9bee63100a7b449d4ae7a6c000b9c")
13   (:|_rev| . "1-3497880f631bd2cea12cdbf8c7d7553f")
14   (:|uri| . "test.com/news")
15   (:|title| . "Fishing Season"))

```

```
16  (:|text| . "Fishing season opens tomorrow"))
```

If you are still running the web administration interface you can look at database **news**, see one document that we just created, and inspect the data in the document:



The screenshot shows the CouchDB Admin Web App interface. The browser address bar displays the URL: 127.0.0.1:5984/_utils/document.html?news/7ae9bee63100a7b449d4ae7a6c000b9c. The breadcrumb navigation shows 'Overview > news > 7ae9bee63100a7b449d4ae7a6c000b9c'. Below the navigation, there are buttons for 'Save Document', 'Add Field', 'Upload Attachment...', and 'Delete Document...'. A table displays the document's fields and values:

Field	Value
_id	"7ae9bee63100a7b449d4ae7a6c000b9c"
_rev	"1-3497880f631bd2cea12cdbf8c7d7553f"
text	"Fishing season opens tomorrow"
title	"Fishing Season"
uri	"test.com/news"

At the bottom of the table, there are links for 'Previous Version' and 'Next Version'.

CouchDB Admin Web App

Fetching Documents

CouchDB uses **views** that are pre-computed using map reduce programs written in Javascript. You don't need to look at it right now, but if after experimenting with the examples in this chapter you decide to use CouchDB for your projects then you will need to read the [CouchDB View Documentation](#).

The clouddb documentation has several examples for using the **parenscrip**t library to convert Lisp to Javascript for **views** but I prefer just using Javascript. One reason for this is that I experiment with writing map reduce functions for creating views while inside the CouchDB Admin Web App in Javascript. I find that using **parenscrip**t is an unnecessary extra level of abstraction. The map reduce view functions I need are relatively simple, and I don't mind writing them in Javascript.

The following listing shows a temporary view to return all documents that have an attribute **title**. The Javascript code in the call to **clouddb:ad-hoc-view**

should be all on one line, I manually reformatted it here for readability:

```
1 * (clouddb:ad-hoc-view
2   "function(doc) {
3     if (doc.title) {
4       emit(null,doc.title)
5     }
6   }")
7
8 ((:|total_rows| . 1) (:|offset| . 0)
9  (:|rows|
10  ((:|id| . "7ae9bee63100a7b449d4ae7a6c000b9c") (:|key|)
11  (:|value| . "Fishing Season"))))
```

You can also match on substrings. The following call to **clouddb:ad-hoc-view** should be all on one line (once again, I manually reformatted it here for readability):

```
1 * (clouddb:ad-hoc-view
2   "function(doc) {
3     if (doc.title.indexOf('Season') != -1) {
4       emit(null,doc.title)
5     }
6   }")
7
8 ((:|total_rows| . 1) (:|offset| . 0)
9  (:|rows|
10  ((:|id| . "7ae9bee63100a7b449d4ae7a6c000b9c")
11  (:|key|)
12  (:|value| . "Fishing Season"))))
```

Please note that ad-hoc views are very inefficient but useful during development. In production you will create named views for specific queries. The CouchDB service will keep the view up to date for you automatically as documents are created, modified, and removed. The following listing shows a “design” document that is used to define a view:

```
1 {
```

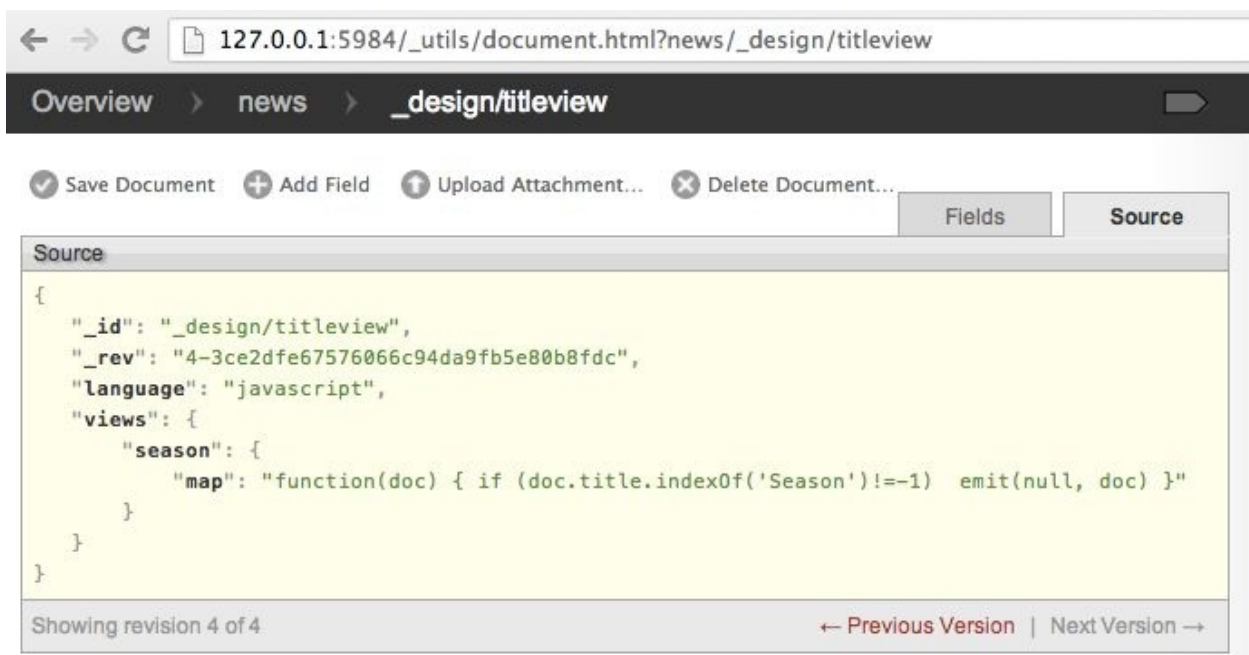
```

2   "_id": "_design/titleview",
3   "_rev": "4-3ce2dfe67576066c94da9fb5e80b8fdc",
4   "language": "javascript",
5   "views": {
6     "season": {
7       "map":
8       "function(doc) {
9         if (doc.title.indexOf('Season')!==-1) emit(null, doc)
10      }"
11   }
12 }
13 }

```

Note the form of the attribute `_id` which is important: the string “`_design/`” followed by the name of the view. In general, you might define multiple views in a single “design” document, but I only defined one here named “season.” Also note that the “season” view only has a map function, not a reduce function. The pattern of using only a map function is common when you want to select a set of documents based on some selection criteria. Using map and reduce functions is useful when you are aggregating data from many documents together.

The following figure shows the CouchDB admin web app while I was editing this view:



You can run this view using:

```

1 * (clouddb:invoke-view "titleview" "season")
2
3 ( (:|total_rows| . 1) (:|offset| . 0)
4   (:|rows|
5     ( (:|id| . "7ae9bee63100a7b449d4ae7a6c000b9c") (:|key|)
6       (:|value| (:|_id| . "7ae9bee63100a7b449d4ae7a6c000b9c")
7         (:|_rev| . "1-3497880f631bd2cea12cdbf8c7d7553f")
8         (:|uri| . "test.com/news")
9         (:|title| . "Fishing Season")
10        (:|text| . "Fishing season opens tomorrow")))))

```

The call to function **clouddb:invoke-view** on line 1 references the name of the design documents without the prefix “_design/” and the name of the view “season” that was defined in this “design” document.

Accessing Document Attributes

If we add an additional test document with “Season” in its title and re-run the view from the last section, we now have two matching results when calling the function **clouddb:invoke-view**:

```

1 * (defvar *results* (clouddb:invoke-view "titleview"
2   "season"))
3
4 *RESULTS*
5
6 * *results*
7
8 ( (:|total_rows| . 2) (:|offset| . 0)
9   (:|rows|
10     ( (:|id| . "7ae9bee63100a7b449d4ae7a6c000b9c") (:|key|)
11       (:|value| (:|_id| . "7ae9bee63100a7b449d4ae7a6c000b9c")
12         (:|_rev| . "1-3497880f631bd2cea12cdbf8c7d7553f")
13         (:|uri| . "test.com/news")
14         (:|title| . "Fishing Season")
15         (:|text| . "Fishing season opens tomorrow"))))
16     ( (:|id| . "7ae9bee63100a7b449d4ae7a6c0019c6") (:|key|)

```

```

15      (:|value| (:|_id| . "7ae9bee63100a7b449d4ae7a6c0019c6")
16      (:|_rev| . "1-1c213a425457a470cad1892e5cf10c1d")
17      (:|uri| . "test.com/news")
18      (:|title| . "Fishing Season")
19      (:|text| . "Fishing season opens tomorrow 3")))))
20 * (cdaddr *results*)
21
22 (((:|id| . "7ae9bee63100a7b449d4ae7a6c000b9c") (:|key|)
23      (:|value| (:|_id| . "7ae9bee63100a7b449d4ae7a6c000b9c")
24      (:|_rev| . "1-3497880f631bd2cea12cdbf8c7d7553f")
25      (:|uri| . "test.com/news")
26      (:|title| . "Fishing Season")
27      (:|text| . "Fishing season opens tomorrow"))))
28 ((:|id| . "7ae9bee63100a7b449d4ae7a6c0019c6") (:|key|)
29      (:|value| (:|_id| . "7ae9bee63100a7b449d4ae7a6c0019c6")
30      (:|_rev| . "1-1c213a425457a470cad1892e5cf10c1d")
31      (:|uri| . "test.com/news")
32      (:|title| . "Fishing Season")
33      (:|text| . "Fishing season opens tomorrow 3")))))
34 * (dolist (r (cdaddr *results*))
35      (print r)
36      (terpri))
37
38 ((:|id| . "7ae9bee63100a7b449d4ae7a6c000b9c") (:|key|)
39      (:|value| (:|_id| . "7ae9bee63100a7b449d4ae7a6c000b9c")
40      (:|_rev| . "1-3497880f631bd2cea12cdbf8c7d7553f")
41      (:|uri| . "test.com/news")
42      (:|title| . "Fishing Season")
43      (:|text| . "Fishing season opens tomorrow"))))
44
45 ((:|id| . "7ae9bee63100a7b449d4ae7a6c0019c6") (:|key|)
46      (:|value| (:|_id| . "7ae9bee63100a7b449d4ae7a6c0019c6")
47      (:|_rev| . "1-1c213a425457a470cad1892e5cf10c1d")
48      (:|uri| . "test.com/news")
49      (:|title| . "Fishing Season")
50      (:|text| . "Fishing season opens tomorrow 3"))))
51 NIL
52 * (defun print-doc (doc) (print (cdddr (caddr doc))))
53

```

```

54 PRINT-DOC
55 * (dolist (r (cdaddr *results*)) (print-doc r) (terpri))
56
57 ( (:|uri| . "test.com/news")
58   (:|title| . "Fishing Season")
59   (:|text| . "Fishing season opens tomorrow"))
60
61 ( (:|uri| . "test.com/news")
62   (:|title| . "Fishing Season")
63   (:|text| . "Fishing season opens tomorrow 3"))
64 NIL
65 * (defun print-doc (doc)
66     (let ((x (cdddr (caddr doc))))
67       (print
68         (list
69          "title:"
70          (cdadr x) "text:" (cdaddr x)))))
71 STYLE-WARNING: redefining COMMON-LISP-USER::PRINT-DOC in DEFUN
72
73 PRINT-DOC
74 * (dolist (r (cdaddr *results*))
75     (print-doc r)
76     (terpri))
77
78 ("title:" "Fishing Season" "text:" "Fishing season opens
79 tomorrow")
80 ("title:" "Fishing Season" "text:" "Fishing season opens
81 tomorrow 3")
82 NIL

```

A Common Lisp Solr Client

The Lucene project is one of the most widely used Apache Foundation projects. Lucene is a flexible library for preprocessing and indexing text, and searching text. I have personally used Lucene on so many projects that it would be difficult counting them. The [Apache Solr Project](#) adds a network interface to the Lucene text indexer and search engine. Solr also adds other utility features to Lucene:

- While Lucene is a library to be embedded in your programs, Solr is a complete system.
- Solr provides good defaults for preprocessing and indexing text and also provides rich support for managing structured data.
- Provides both XML and JSON APIs using HTTP and REST.
- Supports faceted search, geospatial search, and provides utilities for highlighting search terms in surrounding text of search results.
- If your system ever grows to a very large number of users, Solr supports scaling via replication.

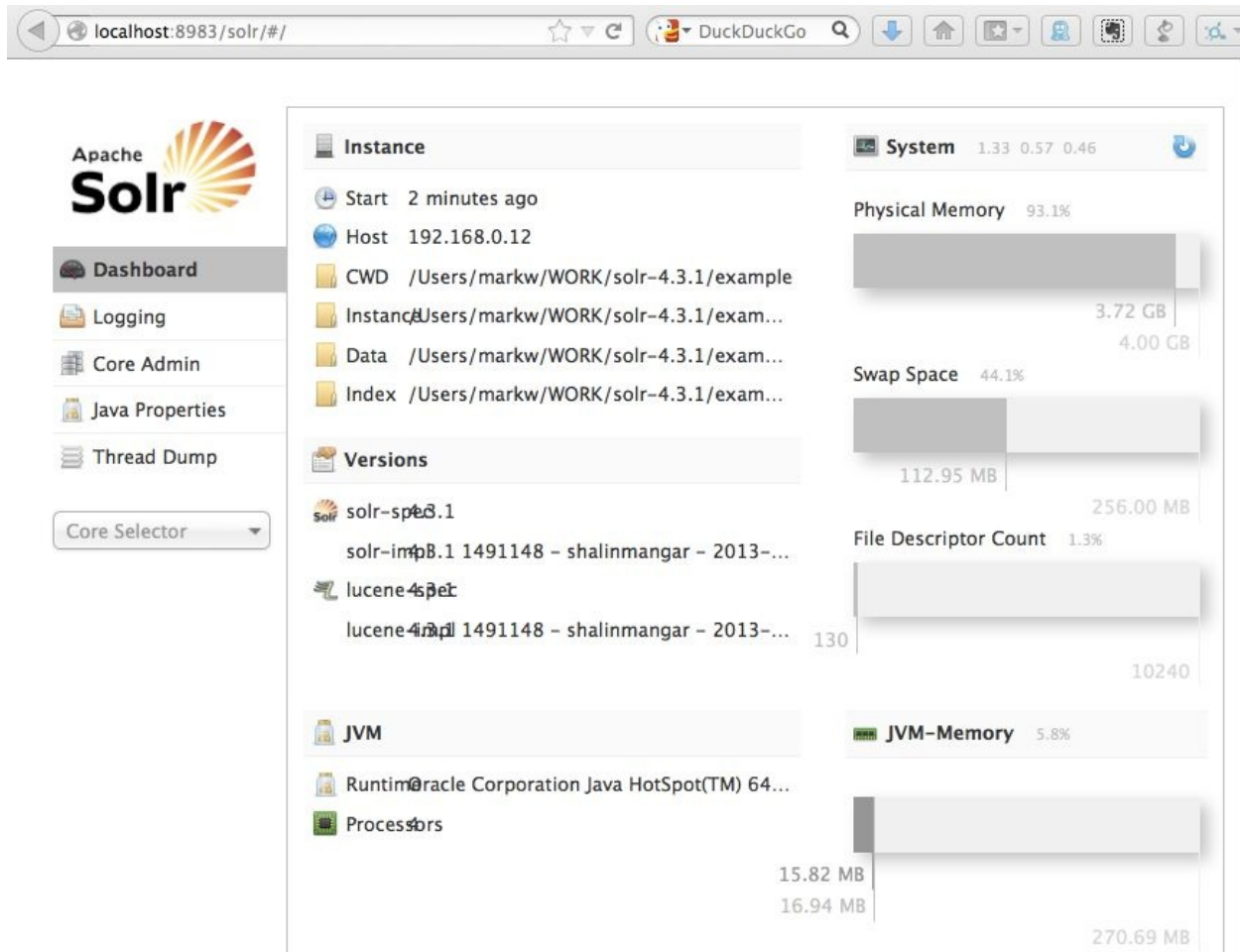
I hope that you will find the Common Lisp example Solr client code in the following sections helps you make Solr part of large systems that you write using Common Lisp.

Installing Solr

Download a [binary Solr distribution](#) and un-tar or un-zip this Solr distribution, cd to the distribution directory, then cd to the example directory and run:

```
1 ~/solr/example> java -jar start.jar
```

You can access the Solr Admin Web App at <http://localhost:8983/solr/#/>. This web app can be seen in the following screen shot:



Solr Admin Web App

There is no data in the Solr example index yet, so following the Solr tutorial instructions:

```
1 ~/> cd ~/solr/example/exampldocs
2 ~/solr/example/exampldocs> java -jar post.jar *.xml
3 SimplePostTool version 1.5
4 Posting files to base url http://localhost:8983/solr/update
5 using content-type application/xml..
6 POSTing file gb18030-example.xml
7 POSTing file hd.xml
8 POSTing file ipod_other.xml
9 POSTing file ipod_video.xml
10 POSTing file manufacturers.xml
11 POSTing file mem.xml
12 POSTing file money.xml
13 POSTing file monitor.xml
```

```
14 POSTing file monitor2.xml
15 POSTing file mp500.xml
16 POSTing file sd500.xml
17 POSTing file solr.xml
18 POSTing file utf8-example.xml
19 POSTing file vidcard.xml
20 14 files indexed.
21 COMMITting Solr index changes
22           to http://localhost:8983/solr/update..
23 Time spent: 0:00:00.480
```

You will learn how to add documents to Solr directly in your Common Lisp programs in a later section.

Assuming that you have a fast Internet connection so downloading Solr was quick, you have hopefully spent less than five or six minutes getting Solr installed and running with enough example search data for the Common Lisp client examples we will play with. Solr is a great tool for storing, indexing, and searching data. I recommend that you put off reading the official Solr documentation for now and instead work through the Common Lisp examples in the next two sections. Later, if you want to use Solr then you will need to carefully read the Solr documentation.

Solr's REST Interface

The [Solr REST Interface Documentation](#) documents how to perform search using HTTP GET requests. All we need to do is implement this in Common Lisp which you will see is easy.

Assuming that you have Solr running and the example data loaded, we can try searching for documents with, for example, the word “British” using the URL <http://localhost:8983/solr/select?q=British>. This is a REST request URL and you can use utilities like **curl** or **wget** to fetch the XML data. I fetched the data in a web browser, as seen in the following screen shot of a Firefox web browser (I like the way Firefox formats and displays XML data):

This XML file does not appear to have any style information associated with it.

```
- <response>
  - <lst name="responseHeader">
    <int name="status">0</int>
    <int name="QTime">3</int>
    - <lst name="params">
      <str name="q">British</str>
    </lst>
  </lst>
- <result name="response" numFound="1" start="0">
  - <doc>
    <str name="id">GBP</str>
    <str name="name">One British Pound</str>
    <str name="manu">U.K.</str>
    <str name="manu_id_s">uk</str>
    - <arr name="cat">
      <str>currency</str>
    </arr>
    - <arr name="features">
      <str>Coins and notes</str>
    </arr>
    <str name="price_c">1,GBP</str>
    <bool name="inStock">true</bool>
    <long name="_version_">1440194917628379136</long>
  </doc>
</result>
</response>
```

Solr Search Results as XML Data

The attributes in the returned search results need some explanation. We indexed several example XML data files, one of which contained the following XML element that we just saw as a search result:

```
1 <doc>
2   <field name="id">GBP</field>
3   <field name="name">One British Pound</field>
4   <field name="manu">U.K.</field>
5   <field name="manu_id_s">uk</field>
6   <field name="cat">currency</field>
7   <field name="features">Coins and notes</field>
8   <field name="price_c">1,GBP</field>
```

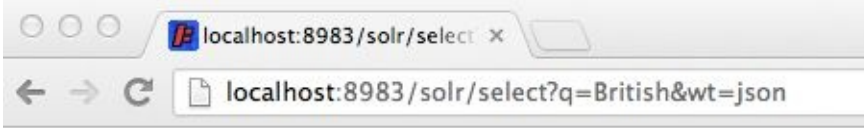
```
9     <field name="inStock">true</field>
10 </doc>
```

So, the search result has the same attributes as the structured XML data that was added to the Solr search index. Solr's capability for indexing structured data is a superset of just indexing plain text. As an example, if we were indexing news stories, then example input data might look like:

```
1 <doc>
2   <field name="id">new_story_0001</field>
3   <field name="title">Fishing Season Opens</field>
4   <field name="text">Fishing season opens on Friday in
Oak Creek.</field>
5 </doc>
```

With this example, a search result that returned this document as a result would return attributes **id**, **title**, and **text**, and the values of these three attributes.

By default the Solr web service returns XML data as seen in the last screen shot. For our examples, I prefer using JSON so we are going to always add a request parameter **wt=json** to all REST calls. The following screen shot shows the same data returned in JSON serialization format instead of XML format of a Chrome web browser (I like the way Chrome formats and displays JSON data with the JSONView Chrome Browser extension):



The screenshot shows a web browser window with the address bar displaying `localhost:8983/solr/select` and the search bar containing `localhost:8983/solr/select?q=British&wt=json`. The main content area displays the JSON response from the Solr search.

```
{
  - responseHeader: {
    status: 0,
    QTime: 1,
    - params: {
      q: "British",
      wt: "json"
    }
  },
  - response: {
    numFound: 1,
    start: 0,
    - docs: [
      - {
        id: "GBP",
        name: "One British Pound",
        manu: "U.K.",
        manu_id_s: "uk",
        - cat: [
          "currency"
        ],
        - features: [
          "Coins and notes"
        ],
        price_c: "1,GBP",
        inStock: true,
        _version_: 1440194917628379100
      }
    ]
  }
}
```

Solr Search Results as JSON Data

You can read the full JSON REST Solr documentation later, but for our use here we will use the following search patterns:

- `http://localhost:8983/solr/select?q=British+One&wt=json` - search for documents with either of the words “British” or “one” in them. Note that in URIs that the “+” character is used to encode a space character. If you wanted a “+” character you would encode it with “%2B” and a space character is encoded as “%20”. The default Solr search option is an OR of the search terms, unlike for example Google Search.
- `http://localhost:8983/solr/select?q=British+AND+one&wt=json` - search for documents that contain both of the words “British” and “one” in them. The search term in plain text is “British AND one”.

Common Lisp Solr Client for Search

As we saw in the earlier in [Network Programming](#) it is fairly simple to use the **drakma** and **cl-json** Common Lisp libraries to call REST services that return JSON data. The function **do-search** defined in the next listing (all the Solr example code is in the file **src/solr-client.lisp**) constructs a query URI as we saw in the last section and uses the **Drackma** library to perform an HTTP GET operation and the **cl-json library** to parse the returned string containing JSON data into Lisp data structures:

```
1 (ql:quickload :drakma)
2 (ql:quickload :cl-json)
3
4 (defun do-search (&rest terms)
5   (let ((query-string (format nil "{A~^+AND+~}" terms)))
6     (cl-json:decode-json-from-string
7       (drakma:http-request
8         (concatenate
9           'string
10            "http://localhost:8983/solr/select?q="
11            query-string
12            "&wt=json")))))
```

This example code does return the search results as Lisp list data; for example:

```
1 * (do-search "British" "one")
2
3 ( (:RESPONSE-HEADER (:STATUS . 0) (:*Q-TIME . 1)
4   (:PARAMS (:Q . "British+AND+one") (:WT . "json")))
5   (:RESPONSE (:NUM-FOUND . 6) (:START . 0)
6     (:DOCS
7       ((:ID . "GBP") (:NAME . "One British Pound") (:MANU .
8         "U.K.")
9         (:MANU--ID--S . "uk") (:CAT "currency")
10        (:FEATURES "Coins and notes")
11        (:PRICE--C . "1,GBP") (:IN-STOCK . T)
12        ( :--VERSION-- . 1440194917628379136))
13       ((:ID . "USD") (:NAME . "One Dollar")
14         (:MANU . "Bank of America")
15         (:MANU--ID--S . "boa") (:CAT "currency")
16         (:FEATURES "Coins and notes"))
```

```

16      (:PRICE--C . "1,USD") (:IN-STOCK . T)
17      (:--VERSION-- . 1440194917624184832))
18      ((:ID . "EUR") (:NAME . "One Euro")
19      (:MANU . "European Union")
20      (:MANU--ID--S . "eu") (:CAT "currency")
21      (:FEATURES "Coins and notes")
22      (:PRICE--C . "1,EUR") (:IN-STOCK . T)
23      (:--VERSION-- . 1440194917626281984))
24      ((:ID . "NOK") (:NAME . "One Krone")
25      (:MANU . "Bank of Norway")
26      (:MANU--ID--S . "nor") (:CAT "currency")
27      (:FEATURES "Coins and notes")
28      (:PRICE--C . "1,NOK") (:IN-STOCK . T)
29      (:--VERSION-- . 1440194917631524864))
30      ((:ID . "0579B002")
31      (:NAME . "Canon PIXMA MP500 All-In-One Photo Printer")
32      (:MANU . "Canon Inc.")
33      (:MANU--ID--S . "canon")
34      (:CAT "electronics" "multifunction printer"
35      "printer" "scanner" "copier")
36      (:FEATURES "Multifunction ink-jet color photo printer"
37      "Flatbed scanner, optical scan resolution of 1,200 x 2,400
dpi"
38      "2.5\" color LCD preview screen" "Duplex Copying"
39      "Printing speed up to 29ppm black, 19ppm color" "Hi-Speed
USB"
40      "memory card: CompactFlash, Micro Drive, SmartMedia,
41      Memory Stick, Memory Stick Pro, SD Card, and
MultiMediaCard")
42      (:WEIGHT . 352.0) (:PRICE . 179.99)
43      (:PRICE--C . "179.99,USD")
44      (:POPULARITY . 6) (:IN-STOCK . T)
45      (:STORE . "45.19214,-93.89941")
46      (:--VERSION-- . 1440194917651447808))
47      ((:ID . "SOLR1000")
48      (:NAME . "Solr, the Enterprise Search Server")
49      (:MANU . "Apache Software Foundation")
50      (:CAT "software" "search")
51      (:FEATURES "Advanced Full-Text Search Capabilities using

```


Lucene"

```
52      "Optimized for High Volume Web Traffic"
53      "Standards Based Open Interfaces - XML and HTTP"
54      "Comprehensive HTML Administration Interfaces"
55      "Scalability - Efficient Replication to other Solr Search
Servers"
56      "Flexible and Adaptable with XML configuration and Schema"
57      "Good unicode support: hÃ©llo (hello with an accent over
the e)")
58      (:PRICE . 0.0) (:PRICE--C . "0,USD") (:POPULARITY . 10)
(:IN-STOCK . T)
59      (:INCUBATIONDATE--DT . "2006-01-17T00:00:00Z")
60      (:--VERSION-- . 1440194917671370752))))))
```

I might modify the search function to return just the fetched documents as a list, discarding the returned Solr meta data:

```
1  * (cdr (caddrr (cadr (do-search "British" "one"))))
2
3  (((:ID . "GBP") (:NAME . "One British Pound") (:MANU . "U.K.")
4    (:MANU--ID--S . "uk") (:CAT "currency") (:FEATURES "Coins
and notes")
5    (:PRICE--C . "1,GBP") (:IN-STOCK . T)
6    (:--VERSION-- . 1440194917628379136))
7  ((:ID . "USD") (:NAME . "One Dollar") (:MANU . "Bank of
America")
8    (:MANU--ID--S . "boa") (:CAT "currency") (:FEATURES "Coins
and notes")
9    (:PRICE--C . "1,USD") (:IN-STOCK . T)
10   (:--VERSION-- . 1440194917624184832))
11  ((:ID . "EUR") (:NAME . "One Euro") (:MANU . "European
Union")
12    (:MANU--ID--S . "eu") (:CAT "currency") (:FEATURES "Coins
and notes")
13    (:PRICE--C . "1,EUR") (:IN-STOCK . T)
14    (:--VERSION-- . 1440194917626281984))
15  ((:ID . "NOK") (:NAME . "One Krone") (:MANU . "Bank of
Norway"))
```

```

16  (:MANU--ID--S . "nor") (:CAT "currency")
17  (:FEATURES "Coins and notes")
18  (:PRICE--C . "1,NOK") (:IN-STOCK . T)
19  (:--VERSION-- . 1440194917631524864))
20  ((:ID . "0579B002")
21   (:NAME . "Canon PIXMA MP500 All-In-One Photo Printer")
22   (:MANU . "Canon Inc.") (:MANU--ID--S . "canon")
23   (:CAT "electronics" "multifunction printer" "printer"
24    "scanner" "copier")
25   (:FEATURES "Multifunction ink-jet color photo printer"
26    "Flatbed scanner, optical scan resolution of 1,200 x 2,400
27    dpi"
28    "2.5\" color LCD preview screen" "Duplex Copying"
29    "Printing speed up to 29ppm black, 19ppm color" "Hi-Speed
30    USB"
31    "memory card: CompactFlash, Micro Drive, SmartMedia, Memory
32    Stick,
33    Memory Stick Pro, SD Card, and MultiMediaCard")
34   (:WEIGHT . 352.0) (:PRICE . 179.99) (:PRICE--C .
35    "179.99,USD")
36   (:POPULARITY . 6) (:IN-STOCK . T) (:STORE .
37    "45.19214,-93.89941")
38   (:--VERSION-- . 1440194917651447808))
39  ((:ID . "SOLR1000") (:NAME . "Solr, the Enterprise Search
40    Server")
41   (:MANU . "Apache Software Foundation") (:CAT "software"
42    "search")
43   (:FEATURES "Advanced Full-Text Search Capabilities using
44    Lucene"
45    "Optimized for High Volume Web Traffic"
46    "Standards Based Open Interfaces - XML and HTTP"
47    "Comprehensive HTML Administration Interfaces"
48    "Scalability - Efficient Replication to other Solr Search
49    Servers"
50    "Flexible and Adaptable with XML configuration and Schema"
51    "Good unicode support: hÃ©llo (hello with an accent over the
52    e)")
53   (:PRICE . 0.0) (:PRICE--C . "0,USD") (:POPULARITY . 10)
54   (:IN-STOCK . T)

```

```
44  (:INCUBATIONDATE--DT . "2006-01-17T00:00:00Z")
45  (:--VERSION-- . 1440194917671370752)))
```

There are a few more important details if you want to add Solr search to your Common Lisp applications. When there are many search results you might want to fetch a limited number of results and then “page” through them. The following strings can be added to the end of a search query:

- &rows=2 this example returns a maximum of two “rows” or two query results.
- &start=4 this example skips the first 4 available results

A query that combines skipping results and limiting the number of returned results looks like this:

```
1 http://localhost:8983/solr/select?
q=British+One&wt=json&start=2&rows=2
```

Common Lisp Solr Client for Adding Documents

In the last example we relied on adding example documents to the Solr search index using the directions for setting up a new Solr installation. In a real application, in addition to performing search requests for indexed documents you will need to add new documents from your Lisp applications. Using the Drakma we will see that it is very easy to add documents.

We need to construct a bit of XML containing new documents in the form:

```
1 <add>
2   <doc>
3     <field name="id">123456</field>
4     <field name="title">Fishing Season</field>
5   </doc>
6 </add>
```

You can specify whatever field names (attributes) that are required for your application. You can also pass multiple <doc></doc> elements in one add request. We will want to specify documents in a Lisp like way: a list of cons values where each cons value is a field name and a value. For the last XML

document example we would like an API that lets us just deal with Lisp data like:

```
1 (do-add '(("id" . "12345")
2         ("title" . "Fishing Season")))
```

One thing to note: the attribute names and values must be passed as strings. Other data types like integers, floating point numbers, structs, *etc.* will not work.

This is nicer than having to use XML, right? The first thing we need is a function to convert a list of cons values to XML. I could have used the XML Builder functionality in the **cxml** library that is available via Quicklisp, but for something this simple I just wrote it in pure Common Lisp with no other dependencies (also in the example file **src/solr-client.lisp**) :

```
1 (defun keys-values-to-xml-string (keys-values-list)
2   (with-output-to-string (stream)
3     (format stream "<add><doc>")
4     (dolist (kv keys-values-list)
5       (format stream "<field name=\"")
6       (format stream (car kv))
7       (format stream "\">")
8       (format stream (cdr kv))
9       (format stream "\"</field>"))
10    (format stream "</doc></add>")))
```

The macro **with-output-to-string** on line 2 of the listing is my favorite way to generate strings. Everything written to the variable **stream** inside the macro call is appended to a string; this string is the return value of the macro.

The following function adds documents to the Solr document input queue but does not actually index them:

```
1 (defun do-add (keys-values-list)
2   (drakma:http-request
3     "http://localhost:8983/solr/update"
4     :method :post
5     :content-type "application/xml"
6     :content ( keys-values-to-xml-string  keys-values-list)))
```

You have noticed in line 3 that I am accessing a Solr server running on **localhost** and not a remote server. In an application using a remote Solr server you would need to modify this to reference your server; for example:

```
1 "http://solr.knowledgebooks.com:8983/solr/update"
```

For efficiency Solr does not immediately add new documents to the index until you commit the additions. The following function should be called after you are done adding documents to actually add them to the index:

```
1 (defun commit-adds ()
2   (drakma:http-request
3     "http://localhost:8983/solr/update"
4     :method :post
5     :content-type "application/xml"
6     :content "<commit></commit>"))
```

Notice that all we need is an empty element **<commit></commit>** that signals the Solr server that it should index all recently added documents. The following repl listing shows everything working together (I am assuming that the contents of the file **src/solr-client.lisp** has been loaded); not all of the output is shown in this listing:

```
1 * (do-add '(("id" . "12345") ("title" . "Fishing Season")))
2
3 200
4 ( (:CONTENT-TYPE . "application/xml; charset=UTF-8")
5   (:CONNECTION . "close"))
6 #<PURI:URI http://localhost:8983/solr/update>
7 #<FLEXI-STREAMS:FLEXI-IO-STREAM {1009193133}>
8 T
9 "OK"
10 * (commit-adds)
11
12 200
13 ( (:CONTENT-TYPE . "application/xml; charset=UTF-8")
14   (:CONNECTION . "close"))
15 #<PURI:URI http://localhost:8983/solr/update>
```

```

16 #<FLEXI-STREAMS:FLEXI-IO-STREAM {10031F20B3}>
17 T
18 "OK"
19 * (do-search "fishing")
20
21 (:RESPONSE-HEADER (:STATUS . 0) (:*Q-TIME . 2)
22   (:PARAMS (:Q . "fishing") (:WT . "json")))
23 (:RESPONSE (:NUM-FOUND . 1) (:START . 0)
24   (:DOCS
25     ((:ID . "12345\"") (:TITLE "Fishing Season\""))
26     (:--VERSION-- . 1440293991717273600))))
27 *

```

Common Lisp Solr Client Wrap Up

Solr has a lot of useful features that we have not used here like supporting faceted search (drilling down in previous search results), geolocation search, and looking up indexed documents by attribute. In the examples I have shown you, all text fields are indexed but Solr optionally allows you fine control over indexing, spelling correction, word stemming, *etc.*

Solr is a very capable tool for storing, indexing, and searching data. I have seen Solr used effectively on projects as a replacement for a relational database or other NoSQL data stores like CouchDB or MongoDB. There is a higher overhead for modifying or removing data in Solr so for applications that involve frequent modifications to stored data Solr might not be a good choice.

NoSQL Wrapup

There are more convenient languages than Common Lisp to use for accessing MongoDB and CouchDB. To be honest, my favorites are Ruby and Clojure. That said, for applications where the advantages of Common Lisp are compelling, it is good to know that your Common Lisp applications can play nicely with MongoDB and CouchDB.

I am a polyglot programmer: I like to use the best programming language for any specific job. When we design and build systems with more than one programming language, there are several options to share data:

- Use foreign function interfaces to call one language from another from

inside one process.

- Use a service architecture and send requests using REST or SOAP.
- Use shared data stores, like relational databases, MongoDB, CouchDB, and Solr.

Hopefully this chapter and the last chapter will provide most of what you need for the last option.

Natural Language Processing

Natural Language Processing (NLP) is the automated processing of natural language text with several goals:

- Determine the parts of speech of words based on the surrounding words.
- Detect if two text documents are similar.
- Categorize text (e.g., is it about the economy, politics, sports, etc.)
- Summarize text
- Determine the sentiment of text
- Detect names (e.g., place names, people's names, product names, etc.)

The example code for this chapter is contained in the file **src/knowledgebooks_nlp.lisp** and needs to load the data in the directory **src/data**. I worked on this Lisp code, and also similar code in Java, from about 2001 to 2011. Starting in 2011 I rewrote these two code bases in the Clojure programming language to support my business. So, the code is **src/knowledgebooks_nlp.lisp** no longer developed or maintained except for removing dead code and doing some simplification for use in this chapter. In any case I hope you find it interesting and useful. I am going to start in the next section with a quick explanation of how to run the example code. If you find the examples interesting then you can also read the rest of this chapter where I explain how the code works.

The approach that I used in my library is now dated. I recommend that you consider taking Andrew Ng's course on Machine Learning on the free online Coursera system and then take one of the Coursera NLP classes for a more modern treatment of NLP.

In addition to the code for my library you might also find the linguistic data in **src/data** useful.

Loading and Running the NLP Library

I repackaged the NLP example code into one long file. The code used to be split over 18 source files. The code should be loaded from the **src** directory:


```

1 % loving-common-lisp git:(master) > cd src
2 % src git:(master) > sbcl
3 * (load "knowledgebooks_nlp.lisp")
4
5 "Starting to load data...."
6 "....done loading data."
7 *

```

Unfortunately, it takes a few minutes to load the required linguistic data so I recommend creating a Lisp image that can be reloaded to avoid the time required to load the data:

```

1 * (sb-ext:save-lisp-and-die "nlp-image" :purify t)
2 [undoing binding stack and other enclosing state... done]
3 [saving current Lisp image into nlp-image:
4 writing 5280 bytes from the read-only space at 0x0x20000000
5 writing 3088 bytes from the static space at 0x0x20100000
6 writing 80052224 bytes from the dynamic space at 0x0x1000000000
7 done]
8 % src git:(master) > ls -lh nlp-image
9 -rw-r--r--  1 markw  staff    76M Jul 13 12:49 nlp-image

```

In line 1 in this repl listing, I use the SBCL built-in function **save-lisp-and-die** to create the Lisp image file. Using **save-lisp-and-die** is a great technique to use whenever it takes a while to set up your work environment. Saving a Lisp image for use the next time you work on a Common Lisp project is reminiscent of working in Smalltalk where your work is saved between sessions in an image file.

You can now start SBCL with the NLP library and data preloaded using the Lisp image that you just created:

```

1 % src git:(master) > sbcl --core nlp-image
2 * (in-package :kbnlp)
3
4 #<PACKAGE "KBNLP">
5 * (defvar
6     *x*

```

```

7      (make-text-object
8        "President Bob Smith talked to Congress about the
economy and taxes"))
9
10     *X*
11     * *X*
12
13     #S(TEXT
14       :URL ""
15       :TITLE ""
16       :SUMMARY "<no summary>"
17       :CATEGORY-TAGS (("news_politics.txt" 0.01648)
18                       ("news_economy.txt" 0.01601))
19       :KEY-WORDS NIL
20       :KEY-PHRASES NIL
21       :HUMAN-NAMES ("President Bob Smith")
22       :PLACE-NAMES NIL
23       :TEXT #("President" "Bob" "Smith" "talked" "to"
24               "Congress" "about" "the"
25               "economy" "and" "taxes")
26       :TAGS #("NNP" "NNP" "NNP" "VBD" "TO" "NNP" "IN" "DT"
27               "NN" "CC" "NNS")
28       :STEMS #("presid" "bob" "smith" "talk" "to" "congress"
29               "about" "the"
30               "economy" "and" "tax"))
31     *

```

At the end of the file `src/` in comments is some test code that processes much more text so that a summary is also generated; here is a bit of the output you will see if you load the test code into your repl:

```

1  :SUMMARY
2  "Often those amendments are an effort to change government
policy
3  by adding or subtracting money for carrying it out. The
initial
4  surge in foreclosures in 2007 and 2008 was tied to
subprime

```

```

5      mortgages issued during the housing boom to people with
shaky
6      credit. 2 trillion in annual appropriations bills for
funding
7      most government programs — usually low profile legislation
that
8      typically dominates the work of the House in June and
July.
9      Bill Clinton said that banking in Europe is a good
business.
10     These days homeowners who got fixed rate prime mortgages
because
11     they had good credit can not make their payments because
they are
12     out of work. The question is whether or not the US dollar
remains
13     the world s reserve currency if not the US economy will
face
14     a depression."
15 :CATEGORY-TAGS (("news_politics.txt" 0.38268)
16                  ("news_economy.txt" 0.31182)
17                  ("news_war.txt" 0.20174))
18 :HUMAN-NAMES ("President Bill Clinton")
19 :PLACE-NAMES ("Florida")

```

The top-level function **make-text-object** takes one required argument that can be either a string containing text or an array of strings where each string is a word or punctuation. Function **make-text-object** has two optional keyword parameters: the URL where the text was found and a title.

```

1 (defun make-text-object (words &key (url "") (title ""))
2   (if (typep words 'string) (setq words (words-from-string
words)))
3   (let* ((txt-obj (make-text :text words :url url :title
title)))
4     (setf (text-tags txt-obj) (part-of-speech-tagger words))
5     (setf (text-stems txt-obj) (stem-text txt-obj))
6     ;; note: we must find human and place names before

```

calling

```
7      ;; pronoun-resolution:
8      (let ((names-places (find-names-places txt-obj)))
9          (setf (text-human-names txt-obj) (car names-places))
10         (setf (text-place-names txt-obj) (cadr names-places)))
11      (setf (text-category-tags txt-obj)
12            (mapcar
13              #'(lambda (x)
14                  (list
15                    (car x)
16                    (/ (cadr x) 1000000.0)))
17                (get-word-list-category (text-text txt-obj))))
18      (setf (text-summary txt-obj) (summarize txt-obj))
19      txt-obj))
```

In line 2, we check if this function was called with a string containing text in which case the function **words-from-string** is used to tokenize the text into an array of string tokens. Line two defines the local variable **txt-obj** with the value of a new text object with only three slots (attributes) defined: **text**, **url**, and **title**. Line 4 sets the slot **text-tags** to the part of speech tokens using the function **part-of-speech-tagger**. We use the function **find-names-places** in line 8 to get person and place names and store these values in the text object. In lines 11 through 17 we use the function **get-word-list-category** to set the categories in the text object. In line 18 we similarly use the function **summarize** to calculate a summary of the text and also store it in the text object. We will discuss these NLP helper functions throughout the rest of this chapter.

The function **make-text-object** returns a struct that is defined as:

```
1 (defstruct text
2   url
3   title
4   summary
5   category-tags
6   key-words
7   key-phrases
8   human-names
9   place-names
10  text)
```

```
11 tags
12 stems)
```

Part of Speech Tagging

This tagger is the Common Lisp implementation of my FastTag open source project. I based this project on Eric Brill's PhD thesis (1995). He used machine learning on annotated text to learn tagging rules. I used a subset of the tagging rules that he generated that were most often used when he tested his tagger. I hand coded his rules in Lisp (and Ruby, Java, and Pascal). My tagger is less accurate, but it is fast - thus the name FastTag.

You can find the tagger implementation in the function **part-of-speech-tagger**. We already saw sample output from the tagger in the last section:

```
1 :TEXT #("President" "Bob" "Smith" "talked" "to" "Congress"
"about" "the"
2         "economy" "and" "taxes")
3 :TAGS #("NNP" "NNP" "NNP" "VBD" "TO" "NNP" "IN" "DT" "NN" "CC"
"NNS")
```

The following table shows the meanings of the tags and a few example words:

Tag	Definition	Example words
CC	Coord Conjunctn	and, but, or
NN	Noun, sing. or mass	dog
CD	Cardinal number	one, two
NNS	Noun, plural	dogs, cats
DT	Determiner	the, some
NNP	Proper noun, sing.	Edinburgh
EX	Existential there	there
NNPS	Proper noun, plural	Smiths
FW	Foreign Word	mon dieu
PDT	Predeterminer	all, both
IN	Preposition	of, in, by
POS	Possessive ending	's
JJ	Adjective	big
PP	Personal pronoun	I, you, she

JJR	Adj., comparative	bigger
PP\$	Possessive pronoun	my, one's
JJS	Adj., superlative	biggest
RB	Adverb	quickly
LS	List item marker	1, One
RBR	Adverb, comparative	faster
MD	Modal	can, should
RBS	Adverb, superlative	fastest
RP	Particle	up, off
WP\$	Possessive-Wh	whose
SYM	Symbol	+, %, &
WRB	Wh-adverb	how, where
TO	"to"	to
\$	Dollar sign	\$
UH	Interjection	oh, oops
#	Pound sign	#
VB	verb, base form	eat, run
"	quote	"
VBD	verb, past tense	ate
VBG	verb, gerund	eating
(Left paren	(
VCN	verb, past part	eaten
)	Right paren)
VBP	Verb, present	eat
,	Comma	,
VBZ	Verb, present	eats
.	Sent-final punct	. ! ?
WDT	Wh-determiner	which, that
:	Mid-sent punct.	: ; —
WP	Wh pronoun	who, what

The function **part-of-speech-tagger** loops through all input words and initially assigns the most likely part of speech as specified in the lexicon. Then a subset of Brill's rules are applied. Rules operate on the current word and the previous word.

As an example Common Lisp implementation of a rule. look for words that are

...an example common Lisp implementation of a rule, recognize words that are tagged as common nouns, but end in “ing” so they should be a gerand (verb form):

```
1 ; rule 8: convert a common noun to a present
2 ;           participle verb (i.e., a gerand)
3 (if (equal (search "NN" r) 0)
4     (let ((i (search "ing" w :from-end t)))
5         (if (equal i (- (length w) 3))
6             (setq r "VBG")))))
```

You can find the lexicon data in the file **src/data/FastTagData.lisp**. This file is List code instead of plain data (that in retrospect would be better because it would load faster) and looks like:

```
1 (defvar lex-hash (make-hash-table :test #'equal :size 110000))
2 (setf (gethash "shakeup" lex-hash) (list "NN"))
3 (setf (gethash "Laurance" lex-hash) (list "NNP"))
4 (setf (gethash "expressing" lex-hash) (list "VBG"))
5 (setf (gethash "citybred" lex-hash) (list "JJ"))
6 (setf (gethash "negative" lex-hash) (list "JJ" "NN"))
7 (setf (gethash "investors" lex-hash) (list "NNS" "NNPS"))
8 (setf (gethash "founding" lex-hash) (list "NN" "VBG" "JJ"))
```

I generated this file automatically from lexicon data using a small Ruby script. Notice that words can have more than one possible part of speech. The most common part of speech for a word is the first entry in the lexicon.

Categorizing Text

The code to categorize text is fairly simple using a technique often called “bag of words.” I collected sample text in several different categories and for each category (like politics, sports, etc.) I calculated the evidence or weight that words contribute to supporting a category. For example, the word “president” has a strong weight for the category “politics” but not for the category “sports.” The reason is that the word “president” occurs frequently in articles and books about politics. The data file that contains the word weightings for each category is **src/data/cat-data-tables.lisp**. You can look at this file; here is a very small part of it:

```

1  ;;; Starting topic: news_economy.txt
2
3  (setf *h* (make-hash-table :test #'equal :size 1000))
4
5  (setf (gethash "news" *h*) 3915)
6  (setf (gethash "debt" *h*) 3826)
7  (setf (gethash "money" *h*) 1809)
8  (setf (gethash "work" *h*) 1779)
9  (setf (gethash "business" *h*) 1631)
10 (setf (gethash "tax" *h*) 1572)
11 (setf (gethash "poverty" *h*) 1512)

```

This file was created by a simple Ruby script (not included with the book's example code) that processes a list of sub-directories, one sub-directory per category. The following listing shows the implementation of function **get-word-list-category** that calculates category tags for input text:

```

1  (defun get-word-list-category (words)
2    (let ((x nil)
3          (ss nil)
4          (cat-hash nil)
5          (word nil)
6          (len (length words))
7          (num-categories (length categoryHashtables))
8          (category-score-accumulation-array
9            (make-array num-categories :initial-element 0)))
10
11      (defun list-sort (list-to-sort)
12        ;;(pprint list-to-sort)
13        (sort list-to-sort
14              #'(lambda (list-element-1 list-element-2)
15                  (> (cadr list-element-1) (cadr list-element-
162))))))
16
17      (do ((k 0 (+ k 1)))
18          ((equal k len))
19          (setf word (string-downcase (aref words k)))
20          (do ((i 0 (+ i 1)))

```



```

21         ((equal i num-categories))
22         (setf cat-hash (nth i categoryHashtables))
23         (setf x (gethash word cat-hash))
24         (if x
25             (setf
26                 (aref category-score-accumulation-array i)
27                 (+ x (aref category-score-accumulation-array
28 i))))))
29     (setf ss '())
30     (do ((i 0 (+ i 1)))
31         ((equal i num-categories))
32         (if (> (aref category-score-accumulation-array i) 0.01)
33             (setf
34                 ss
35                 (cons
36                     (list
37                         (nth i categoryNames)
38                         (round (* (aref category-score-accumulation-
39 array i) 10))))
40                     ss))))
41     (setf ss (list-sort ss))
42     (let ((cutoff (/ (cadr ss) 2))
43           (results-array '()))
44         (dolist (hit ss)
45             (if (> (cadr hit) cutoff)
46                 (setf results-array (cons hit results-array))))
47         (reverse results-array)))

```

One thing to notice in this listing is lines 11 through 15 where I define a nested function **list-sort** that takes a list of sublists and sorts the sublists based on the second value (which is a number) in the sublists. I often nest functions when the “inner” functions are only used in the “outer” function.

Lines 2 through 9 define several local variables used in the outer function. The global variable **categoryHashtables** is a list of word weighting score hash tables, one for each category. The local variable **category-score-accumulation-array** is initialized to an array containing the number zero in each element and will be used to “keep score” of each category. The highest scored categories will

be the return value for the outer function.

Lines 17 through 27 are two nested loops. The outer loop is over each word in the input word array. The inner loop is over the number of categories. The logic is simple: for each word check to see if it has a weighting score in each category's word weighting score hash table and if it is, increment the matching category's score.

The local variable `ss` is set to an empty list on line 28 and in the loop in lines 29 through 38 I am copying over categories and their scores when the score is over a threshold value of 0.01. We sort the list in `ss` on line 39 using the inner function and then return the categories with a score greater than the median category score.

Stemming Text

Stemming text is often used for information retrieval tasks. Stemming maps words to the root word yielding tokens that are often not legal words. For example “president” and “presidential” would both map to “presid” so searching for the token “presid” in a stemmed token list would match either word.

The Common Lisp stemming code in the file `src/knowledgebooks_nlp.lisp` was written by Steven M. Haflich and is based on based on the work of Martin Porter.

As we saw in a previous example the effects of stemming:

```
1 :TEXT #("President" "Bob" "Smith" "talked" "to" "Congress"
2         "about" "the"
3         "economy" "and" "taxes")
3 :STEMS #("presid" "bob" "smith" "talk" "to" "congress" "about"
4         "the"
5         "economi" "and" "tax"))
```

The top level function **make-text-object** stems all words in input text.

Detecting People's Names and Place Names

The functions that support identifying people's names and place names in text are:

- `find-names` (`words` `tags` `exclusion-list`) – `words` is an array of strings for the words in text, `tags` are the parts of speech tags (from `FastTag`), and the `exclusion list` is an array of words that you want to exclude from being considered as parts of people's names. The list of found names records starting and stopping indices for names in the array `words`.
- `not-in-list-find-names-helper` (`a-list` `start` `end`) – returns true if a found name is not already been added to a list for saving people's names in text
- `find-places` (`words` `exclusion-list`) – this is similar to `find-names`, but it finds place names. The list of found place names records starting and stopping indices for place names in the array `words`.
- `not-in-list-find-places-helper` (`a-list` `start` `end`) – returns true if a found place name is not already been added to a list for saving place names in text
- `build-list-find-name-helper` (`v` `indices`) – This converts lists of start/stop word indices to strings containing the names
- `find-names-places` (`txt-object`) – this is the top level function that your application will call. It takes a **`destructure text`** object as input and modifies the **`destructure text`** by adding people's and place names it finds in the text. You saw an example of this earlier in this chapter.

I will let you read the code and just list the top level function:

```

1 (defun find-names-places (txt-object)
2   (let* ((words (text-text txt-object))
3         (tags (text-tags txt-object))
4         (place-indices (find-places words nil))
5         (name-indices (find-names words tags place-indices))
6         (name-list
7           (remove-duplicates
8             (build-list-find-name-helper words name-indices)
9             :test #'equal))
10        (place-list
11          (remove-duplicates
12            (build-list-find-name-helper words place-indices)
13            :test #'equal)))
14   (let ((ret '()))
15     (dolist (x name-list)
16       (if (search " " x)
17         (setq ret (cons x ret))))
18     (setq name-list (reverse ret)))

```

```

17      (list
18        (remove-shorter-names name-list)
19        (remove-shorter-names place-list))))

```

In line 2 we are using the slot accessor **text-text** to fetch the array of word tokens from the text object. In lines 3, 4, and 5 we are doing the same for part of speech tags, place name indices in the words array, and person names indices in the words array.

In lines 6 through 11 we are using the function **build-list-find-name-helper** twice to construct the person names and place names as strings given the indices in the words array. We are also using the Common Lisp built-in function **remove-duplicates** to get rid of duplicate names.

In lines 12 through 16 we are discarding any persons names that do not contain a space, that is, only keep names that are at least two word tokens. Lines 17 through 19 define the return value for the function: a list of lists of people and place names using the function **remove-shorter-names** twice to remove shorter versions of the same names from the lists. For example, if we had two names “Mr. John Smith” and “John Smith” then we would want to drop the shorter name “John Smith” from the return list.

Summarizing Text

There are many applications for summarizing text. As an example, if you are writing a document management system you will certainly want to use something like Solr to provide search functionality. Solr will return highlighted matches in snippets of indexed document field values. Using summarization, when you add documents to a Solr (or other) search index you could create a new unindexed field that contains a document summary. Then when the users of your system see search results they will see the type of highlighted matches in snippets they are used to seeing in Google, Bing, or DuckDuckGo search results, and, they will see a summary of the document.

Sounds good? The problem to solve is getting good summaries of text and the technique used may have to be modified depending on the type of text you are trying to summarize. There are two basic techniques for summarization: a practical way that almost everyone uses, and an area of research that I believe has so far seen little practical application. The techniques are sentence extraction and abstraction of text into a shorter form by combining and altering sentences

and abstraction of text into a shorter form by combining and altering sentences. We will use sentence extraction.

How do we choose which sentences in text to extract for the summary? The idea I had in 1999 was simple. Since I usually categorize text in my NLP processing pipeline why not use the words that gave the strongest evidence for categorizing text, and find the sentences with the largest number of these words. As a concrete example, if I categorize text as being “politics”, I identify the words in the text like “president”, “congress”, “election”, *etc.* that triggered the “politics” classification, and find the sentences with the largest concentrations of these words.

Summarization is something that you will probably need to experiment with depending on your application. My old summarization code contained a lot of special cases, blocks of commented out code, *etc.* I have attempted to shorten and simplify my old summarization code for the purposes of this book as much as possible and still maintain useful functionality.

The function for summarizing text is fairly simple because when the function **summarize** is called by the top level NLP library function **make-text-object**, the input text has already been categorized. Remember from the example at the beginning of the chapter that the category data looks like this:

```
1 :CATEGORY-TAGS (("news_politics.txt" 0.38268)
2                  ("news_economy.txt" 0.31182)
3                  ("news_war.txt" 0.20174))
```

This category data is saved in the local variable **cats** on line 4 of the following listing.

```
1 (defun summarize (txt-obj)
2   (let* ((words (text-text txt-obj))
3         (num-words (length words))
4         (cats (text-category-tags txt-obj))
5         (sentence-count 0)
6         (best-sentences sentence (score 0))
7     ;; loop over sentences:
8     (dotimes (i num-words)
9       (let ((word (svref words i)))
10        (dolist (cat cats)
```

```

11         (let* ((hash (gethash (car cat) categoryToHash))
12                (value (gethash word hash)))
13             (if value
14                 (setq score (+ score (* 0.01 value (cadr
cat))))))
15         (push word sentence)
16         (if (or (equal word ".") (equal word "!") (equal
word ";"))
17             (let ()
18                 (setq sentence (reverse sentence))
19                 (setq score (/ score (1+ (length sentence))))
20                 (setq sentence-count (1+ sentence-count))
21                 (format t "%A : A%" sentence score)
22                 ;; process this sentence:
23                 (if (and
24                     (> score 0.4)
25                     (> (length sentence) 4)
26                     (< (length sentence) 30))
27                     (progn
28                         (setq sentence
29                             (reduce
30                                 #'(lambda (x y) (concatenate
'string x " " y))
31                                 (coerce sentence 'list)))
32                         (push (list sentence score) best-
sentences)))
33                     (setf sentence nil score 0))))))
34     (setf
35         best-sentences
36         (sort
37             best-sentences
38             #'(lambda (x y) (> (cadr x) (cadr y)))))
39     (if best-sentences
40         (replace-all
41             (reduce #'(lambda (x y) (concatenate 'string x " "
y))
42                 (mapcar #'(lambda (x) (car x)) best-
sentences))
43             " ." ".")

```

```
"<no summary>" )))
```

The nested loops in lines 8 through 33 look a little complicated, so let's walk through it. Our goal is to calculate an importance score for each word token in the input text and to then select a few sentences containing highly scored words. The outer loop is over the word tokens in the input text. For each word token we loop over the list of categories, looking up the current word in each category hash and incrementing the score for the current word token. As we increment the word token scores we also look for sentence breaks and save sentences.

The complicated bit of code in lines 16 through 32 where I construct sentences and their scores, and store sentences with a score above a threshold value in the list **best-sentences**. After the two nested loops, in lines 34 through 44 we simply sort the sentences by score and select the “best” sentences for the summary. The extracted sentences are no longer in their original order, which can have strange effects, but I like seeing the most relevant sentences first.

Text Mining

Text mining in general refers to finding data in unstructured text. We have covered several text mining techniques in this chapter:

- Named entity recognition - the NLP library covered in this chapter recognizes person and place entity names. I leave it as an exercise for you to extend this library to handle company and product names. You can start by collecting company and product names in the files **src/data/names/names.companies** and **src/data/names/names.products** and extend the library code.
- Categorizing text - you can increase the accuracy of categorization by adding more weighted words/terms that support categories. If you are already using Java in the systems you build, I recommend the Apache OpenNLP library that is more accurate than the simpler “bag of words” approach I used in my Common Lisp NLP library. If you use Python, then I recommend that you also try the NLTK library.
- Summarizing text.

In the next chapter I am going to cover another “data centric” topic: performing information gathering on the web. You will likely find some synergy between being able to use NLP to create structured data from unstructured text.

Information Gathering

This chapter covers information gathering on the web using data sources and general techniques that I have found useful. When I was planning this new book edition I had intended to also cover some basics for using the Semantic Web from Common Lisp, basically distilling some of the data from my previous book “Practical Semantic Web and Linked Data Applications, Common Lisp Edition” published in 2011. However since a [free PDF is now available for that book](#) I decided to just refer you to my previous work if you are interested in the Semantic Web and Linked Data. You can also find the Java edition of this previous book on my web site.

Gathering information from the web in realtime has some real advantages:

- You don’t need to worry about storing data locally.
- Information is up to date (depending on which web data resources you choose to use).

There are also a few things to consider:

- Data on the web may have legal restrictions on its use so be sure to read the terms and conditions on web sites that you would like to use.
- Authorship and validity of data may be questionable.

DBPedia Lookup Service

Wikipedia is a great source of information. As you may know, you can download a [data dump of all Wikipedia data](#) with or without version information and comments. When I want fast access to the entire Wikipedia set of English language articles I choose the second option and just get the current pages with no comments of versioning information. [This is the direct download link for current Wikipedia articles](#). There are no comments or user pages in this GZIP file. This is not as much data as you might think, only about 9 gigabytes compressed or about 42 gigabytes uncompressed.

Wikipedia is a great resource to have on-hand but I am going to show you in this

section how to access the Semantic Web version or Wikipedia, [DBPedia](#) using the DBPedia Lookup Service in the next code listing that shows the contents of the example file **dbpedia-lookup.lisp**:

```
1 (ql:quickload :drakma)
2 (ql:quickload :babel)
3 (ql:quickload :s-xml)
4
5 ;; utility from http://cl-cookbook.sourceforge.net/strings.html#manip:
6 (defun replace-all (string part replacement &key (test
#'char=))
7   "Returns a new string in which all the occurrences of the part
8   is replaced with replacement."
9   (with-output-to-string (out)
10    (loop with part-length = (length part)
11          for old-pos = 0 then (+ pos part-length)
12          for pos = (search part string
13                          :start2 old-pos
14                          :test test)
15          do (write-string string out
16                          :start old-pos
17                          :end (or pos (length string)))
18          when pos do (write-string replacement out)
19          while pos)))
20
21 (defstruct dbpedia-data uri label description)
22
23 (defun dbpedia-lookup (search-string)
24   (let* ((s-str (replace-all search-string " " "+"))
25          (s-uri
26            (concatenate
27              'string
28              "http://lookup.dbpedia.org/api/search.asmx/KeywordSearch?
29              QueryString="
30              s-str))
31          (response-body nil)
32          (response-status nil)
```

```

32         (response-headers nil)
33         (xml nil)
34         ret)
35     (multiple-value-setq
36         (response-body response-status response-headers)
37         (drakma:http-request
38             s-uri
39             :method :get
40             :accept "application/xml"))
41     ;; (print (list "raw response body as XML:" response-
body))
42     ;;(print (list ("status:" response-status "headers:"
response-headers)))
43     (setf xml
44         (s-xml:parse-xml-string
45             (babel:octets-to-string response-body)))
46     (dolist (r (cdr xml))
47         ;; assumption: data is returned in the order:
48         ;;     1. label
49         ;;     2. DBPedia URI for more information
50         ;;     3. description
51         (push
52             (make-dbpedia-data
53                 :uri (cadr (nth 2 r))
54                 :label (cadr (nth 1 r))
55                 :description
56                 (string-trim
57                     '(#\Space #\NewLine #\Tab)
58                     (cadr (nth 3 r))))
59             ret))
60     (reverse ret)))
61
62 ;; (dbpedia-lookup "berlin")

```

I am only capturing the attributes for DBPedia URI, label and description in this example code. If you uncomment line 41 and look at the entire response body from the call to DBPedia Lookup, you can see other attributes that you might want to capture in your applications.

Here is a sample call to the function **dbpedia-lookup** (only some of the returned data is shown):

```
1 * (load "dbpedia-lookup.lisp")
2 * (dbpedia-lookup "berlin")
3
4 (#S(DBPEDIA-DATA
5   :URI "http://dbpedia.org/resource/Berlin"
6   :LABEL "Berlin"
7   :DESCRIPTION
8     "Berlin is the capital city of Germany and one of the 16
states of Germany.
9     With a population of 3.5 million people, Berlin is
Germany's largest city
10    and is the second most populous city proper and the eighth
most populous
11    urban area in the European Union. Located in northeastern
Germany, it is
12    the center of the Berlin-Brandenburg Metropolitan Region,
which has 5.9
13    million residents from over 190 nations. Located in the
European Plains,
14    Berlin is influenced by a temperate seasonal climate.")
15 ...)
```

Wikipedia, and the DBPedia linked data for of Wikipedia are great sources of online data. If you get creative, you will be able to think of ways to modify the systems you build to pull data from DPPedia. One warning: Semantic Web/Linked Data sources on the web are not available 100% of the time. If your business applications depend on having the DBPedia always available then you can follow the instructions on the [DBPedia web site](#) to install the service on one of your own servers.

Web Spiders

When you write web spiders to collect data from the web there are two things to consider:

- Make sure you read the terms of service for web sites whose data you want

to use. I have found that calling or emailing web site owners explaining how I want to use the data on their site usually works to get permission.

- Make sure you don't access a site too quickly. It is polite to wait a second or two between fetching pages and other assets from a web site.

We have already used the Drakma web client library in this book. See the files **src/dbpedia-lookup.lisp** (covered in the last section) and **src/solr-client.lisp** (covered in the [Chapter on NoSQL](#)). Paul Nathan has written library using Drakma to crawl a web site with an example to print out links as they are found. His code is available under the AGPL license at articulate-lisp.com/src/web-trotter.lisp and I recommend that as a starting point. The documentation is [here](#).

I find it is sometimes easier during development to make local copies of a web site so that I don't have to use excess resources from web site hosts. Assuming that you have the **wget** utility installed, you can mirror a site like this:

```
1 wget -m -w 2 http://knowledgebooks.com/
2 wget -mk -w 2 http://knowledgebooks.com/
```

Both of these examples have a two second delay between HTTP requests for resources. The option **-m** indicates to recursively follow all links on the web site. The **-w 2** option delays for two seconds between requests. The option **-mk** converts URI references to local file references on your local mirror. The second example on line 2 is more convenient.

We covered reading from local files in the [Chapter on Input and Output](#). One trick I use is to simply concatenate all web pages into one file. Assuming that you created a local mirror of a web site, cd to the top level directory and use something like this:

```
1 cd knowledgebooks.com
2 cat *.html */*.html > ../web_site.html
```

You can then open the file, search for text in in **p**, **div**, **h1**, etc. HTML elements to process an entire web site as one file.

Using Apache Nutch

[Apache Nutch](#), like Solr, is built on Lucene search technology. I use Nutch as a

“search engine in a box” when I need to spider web sites and I want a local copy with a good search index.

Nutch handles a different developer’s use case over Solr which we covered in the [Chapter on NoSQL](#). As we saw, Solr is an effective tool for indexing and searching structured data as documents. With very little setup, Nutch can be set up to automatically keep an up to date index of a list of web sites, and optionally follow links to some desired depth from these “seed” web sites.

You can use the same Common Lisp client code that we used for Solr with one exception; you will need to change the root URI for the search service to:

```
1  http://localhost:8080/opensearch?query=
```

So the modified client code **src/solr-client.lisp** needs one line changed:

```
1 (defun do-search (&rest terms)
2   (let ((query-string (format nil "{A~^+AND+~}" terms)))
3     (cl-json:decode-json-from-string
4       (drakma:http-request
5         (concatenate
6           'string
7           "http://localhost:8080/opensearch?query="
8           query-string
9           "&wt=json")))))
```

Early versions of Nutch were very simple to install and configure. Later versions of Nutch have been more complex, more performant, and have more services, but it will take you longer to get set up than earlier versions. If you just want to experiment with Nutch, you might want to start with an earlier version.

The [OpenSearch.org](#) web site contains many public OpenSearch services that you might want to try. If you want to modify the example client code in **src/solr-client.lisp** you should start with OpenSearch services that return JSON data and [OpenSearch Community JSON formats web page](#) is a good place to start. Some of the services on this web page like the New York Times service require that you sign up for a developer’s API key.

When I start writing an application that requires web data (no matter which

programming language I am using) I start by finding services that may provide the type of data I need and do my initial development with a web browser with plugin support to nicely format XML and JSON data. I do a lot of exploring and take a lot of notes before I write any code.

Wrap Up

I tried to provide some examples and advice in this short chapter to show you that even though other languages like Ruby and Python have more libraries and tools for gathering information from the web, Common Lisp has good libraries for information gathering also and they are easily used via Quicklisp.

I would like to thank you for reading my book and I hope that you enjoyed it. As I mentioned in the [Introduction](#) I have been using Common Lisp since the mid-1980s, and other Lisp dialects for longer than that. I have always found something almost magical developing in Lisp. Being able to extend the language with macros and using the development technique of building a mini-language in Lisp customized for an application enables programmers to be very efficient in their work. I have usually found that this bottom-up development style helps me deal with software complexity because the lower level functions tend to get well tested while the overall system being developed is not yet too complex to fully understand. Later in the development process these lower level functions and utilities almost become part of the programming language and the higher level application logic is easier to understand because you have fewer lines of code to fit inside your head during development.

I think that unless a programmer works in very constrained application domains, it often makes sense to be a polyglot programmer. I have tried, especially in the new material for this third edition, to give you confidence that Common Lisp is good for both general software development language and also as “glue” to tie different systems together.