# 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 2021-03-28

# Contents

# Cover Material, Copyright, and License



This eBook will be updated occasionally so please periodically check the leanpub.com web page for this book[1] for updates.

This is the seventh edition released spring of 2021.

Please visit the author’s website[2].

If you found a copy of this book on the web and find it of value then please consider buying a copy at leanpub.com/lovinglisp[3] to support the author and fund work for future updates. You can also download a free copy from my website[4].

[1] https://leanpub.com/lovinglisp
[2] http://markwatson.com
[3] https://leanpub.com/lovinglisp
[4] https://markwatson.com/#books

# Preface

## Notes on the Seventh Edition Published March 2021

I added two short chapters to the previous edition: Knowledge Graph Sampler for Creating Small Custom Knowledge Graphs and Using Common Lisp With Wolfram/One.

## Notes on the Sixth Edition Published June 2020

Two examples optionally use the CAPI user interface toolkit provided with LispWorks Common Lisp[5] and work with the free personal edition. The first CAPI application is Knowledge Graph Navigator[6] and the second CAPI example is Knowledge Graph Creator[7]. Both of these examples build up utilities for working with Knowledge Graphs and the Semantic Web.

I expand the Plot Library chapter to generate either PNG graphics files or if you are using the free personal edition of LispWorks you can also direct plotting output to a new window in interactive programs.

I added a new chapter on using the py4cl library to embed Python libraries and application code into a Common Lisp system. I provide new examples for embedding spaCy and TensorFlow applications in Common Lisp applications. In earlier editions, I used a web services interface to wrap Python code using spaCy and TensorFlow. I am leaving that chapter intact, renaming it from "Using Python Deep Learning Models In Common Lisp" to "Using Python Deep Learning Models In Common Lisp With a Web Services Interface." The new chapter for this edition is "Using the PY4CL Library to Embed Python in Common Lisp."

## Notes on the Fifth Edition Published September 2019

There were two chapters added:

- A complete application for processing text to generate data for Knowledge Graphs (targeting the open source Neo4J graph database and also support RDF semantic web/linked data)
- A library for accessing the state of the art spaCy natural language processing (NLP) library and also a state of the art deep learning model. These models are implemented in thin Python wrappers that use Python libraries like spaCy, PyTorch, and TensorFlow. These examples replace a simple hybrid Java and Common Lisp example in previous editions.

[5] https://lispworks.com
[6] http://knowledgegraphnavigator.com
[7] http://kgcreator.com

I have added text and explanations as appropriate throughout the book and I removed the CouchDB examples.

I have made large changes to how the code for this book is packaged. I have reorganized the example code on GitHub by providing the examples as multiple Quicklisp libraries or applications. I now do this with all of my Common Lisp code and it makes it easier to write smaller libraries that can be composed into larger applications. In my own workflow, I also like to use Makefile targets to build standalone applications that can be run on other computers without installing Lisp development environments. Please follow the directions at the end of the Preface for configuring Quicklisp for easy builds and use of the example software for this book.

# Why Use Common Lisp?

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.

# A Request from the Author

I spent time writing this book to help you, dear reader. I release this book under the Creative Commons "share and share alike, no modifications, no commercial reuse" license and set the minimum purchase price to $5.00 in order to reach the most readers. Under this license you can share a PDF version of this book with your friends and coworkers and I encourage you to do so. If you found this book on the web (or it was given to you) and if it provides value to you then please consider doing one of the following to support my future writing efforts and also to support future updates to this book:

- Purchase a copy of this book leanpub.com/lovinglisp/[8] or any other of my leanpub books at https://leanpub.com/u/markwatson[9]
- Hire me as a consultant[10]

I enjoy writing and your support helps me write new editions and updates for my books and to develop new book projects. Thank you!

[8] https://leanpub.com/lovinglisp/
[9] https://leanpub.com/u/markwatson
[10] https://markwatson.com/

# Older Book Editions

The fourth edition of this book was released in May 2017 and the major changes were:

- Added an example application KGCreator that processes text data to automatically generate data for Knowledge Graphs. This example application supports the Neo4J graph database as well as semantic web/linked data systems. The major changes were:
- Added a backpropagation neural network example
- Added a deep learning example using the Java based Armed Bear Common Lisp with the popular DeepLearning4j library
- Added a heuristic search example
- Added two machine learning examples (K-Means clustering and SVM classification) using the CLML library
- A few edits to the previous text

The third edition was released in October 2014. The major changes made in the 2014 edition are:

- I reworked the chapter **Common Lisp Basics**.
- I added material to the chapter on using QuickLisp.

The second edition was released in 2013 and was derived from the version that I distributed on my web site and I moved production of the book to leanpub.com[11].

# Acknowledgments

I would like to thank Jans Aasman[12] for contributing as technical editor for the fourth edition of this book. Jans is CEO of Franz.com[13] which sells Allegro Common Lisp[14] as well as tools for semantic web and linked data applications.

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: Martin Lightheart, Tong-Kiat Tan, Rainer Joswig, Gerold Rupprecht, HN member rurban, David Cortesi. I would like to thanks the following Reddit /r/lisp readers who pointed out

[11] https://leanpub.com/u/markwatson
[12] https://en.wikipedia.org/wiki/Jans_Aasman
[13] http://franz.com/
[14] http://franz.com/products/allegro-common-lisp/

mistakes in the fifth edition of this book: arnulfslayer, rpiirp, and itmuckel. I would like to thank Ted Briscoe for pointing out a problem with the spacy web client example in the 6th edition.

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 and giving me permission to reuse his phrase.

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

# Setting Up Your Common Lisp Development System and Quicklisp

These instructions assume the use of SBCL. See comments for LispWorks, Franz Common Lisp, and Closure Common List at the end of this section. I assume that you have installed SBCL and Quicklisp by following the instructions at lisp-lang.org/learn/getting-started[15]. These instructions also guide you through installing the Slime extensions for Emacs. I use both Emacs + Slime and VSCode with Common Lisp plugins for editing Common Lisp. If you like VSCode then I recommend Yasuhiro Matsumoto's Lisp plugin for syntax highlighting. For both Emacs and VSCode I usually run a separate REPL in a terminal window and don't run an editor-integrated REPL. I think that am in the minority in using a separate REPL running in a shell.

I have been using Common Lisp since about 1982 and Quicklisp has been the most revolutionary change in my Common Lisp development (even more so than getting a hardware Lisp Machine and the availability of Coral Common Lisp on the Macintosh). I am going to ask you, dear reader, to trust me and adopt the following advice that I have adopted from Zach Beane[16], the creator and maintainer of Quicklisp:

- Create the file ~/.config/common-lisp/source-registry.conf.d/projects.conf if it does not exist on your system
- Assuming that you have cloned the repository for this book (loving-common-lisp) in your home directory (if you have a special place where you clone git repos, adjust the following), edit this configuration file to look like this:

```
(:tree
 (:home "loving-common-lisp/src/")
)
```

This will make subdirectories of loving-common-lisp/src/ load-able by using Quicklisp. For example, the subdirectory loving-common-lisp/src/spacy_client contains a package named **spacy** that can now be accessed from any directory on your system using:

[15] https://lisp-lang.org/learn/getting-started/
[16] https://www.xach.com

```
$ sbcl
(ql:quickload "spacy")
* (spacy:spacy-client "My sister has a dog Henry. She loves him.")
* (defvar x (spacy:spacy-client "President Bill Clinton went to Congress. He gave a \
speech on taxes and Mexico."))
* (spacy:spacy-data-entities x)
* (spacy:spacy-data-tokens x)
```

This example uses the deep learning NLP models in spaCy.

# List of Quicklisp Projects and Small Examples in this Book

The major example libraries and applications will be in their own packages. The function and data definitions for all short code snippets in this book are in a package **loving-snippets** in the subdirectory **loving-common-lisp/src/loving_snippets**. Whenever you work through the short examples in this book I will assume that you have opened a SBCL (or other Common Lisp) REPL and loaded this package:

```
$ sbcl
* (ql:quickload "spacy_web_client")
```

On one of my Linux laptops, for reasons I haven't discovered yet, using ~/**.config/common-lisp/source-registry.conf.d/projects.conf** to set a root directory for Quicklisp to look for packages does not work. If by small chance this does not work for you, you can set symbolic file links from the example book packages to your ~/**quicklisp/local-projects** directory. This is the directory where Quicklisp stores local copies of libraries that you install. For example:

```
$ cd ~/quicklisp/local-projects
$ ln -s loving-common-lisp/src/loving_snippets .
$ ln -s loving-common-lisp/src/kgcreator .
$ ln -s loving-common-lisp/src/kbnlp .
```

etc.

Hopefully you won't have to bother doing this workaround.

While most of the longer examples in this book are Quicklisp projects, there are also many very short code snippets in the book that are found in the subdirectories src/code_snippets_for_book and a few short program examples not configured as Quicklisp projects in the src/loving_snippets subdirectory:

```
$ ls code_snippets_for_book
closure1.lisp                       nested.lisp                       read-test-1.lisp
readline-test.lisp do1.lisp                       read-from-string-test.lisp
read-test-2.lisp               recursion1.lisp
Marks-MacBook:src $ ls loving_snippets
HTMLstream.lisp                       astar_search.lisp               macro1.lisp
Hopfield_neural_network.lisp         backprop_neural_network.lisp        macro2.lisp README.md
ambda1.lisp                        mongo_news.lisp
```

The longer examples packaged as Quicklisp projects in the **src** directory are:

- fasttag: my part of speech tagger
- solr_examples: client for open source solr search engine
- categorize_summarize: my NLP code for categorizing text and generating summaries
- coref_web_client: client for a spaCy based web service that performs anaphora resolution (i.e., replaces pronouns in text with the nouns that the pronouns refer to)
- hunchentoot_examples : examples so web services and web clients
- spacy_web_client: client for a general purpose web service using state of the art deep learning models for NLP
- clml_examples: examples using the Common List Machine Learning library
- kbnlp: My NLP code
- myutils: miscelanious functions that are used in several other example libraries in this book
- webscrape: demo for how to scrape web sites
- clsql_examples : examples showing how to access relational databases using the CLSQL library
- entities_dbpedia : use the public DbPedia (data from WikiPedia) public web interface to get information about people, companies, locations, etc.
- kgcreator: my application for processing text, extracting entities, and generating data for Knowledge Graphs (supports Neo4J and RDF semantic web/linked data applications
- kgn: the application Knowledge Graph Navigator[17]
- plotlib: a very simple plotting library that writes plots to PNG graphics files

I have used the SBCL implementation of Common Lisp in this book. There are many fine Common Lisp implementations from Franz, LispWorks, Clozure Common Lisp, etc. If you have any great difficulty adopting the examples to your choice of Common Lisp implementations and performing web search does not suggest a solution then you can reach me through my web site markwatson.com[18]. The examples that may not be portable are creating a standalone executable for my KGCreator example and the examples using the Common Lisp Machine Learning library.

[17] http://knowledgegraphnavigator.com
[18] https://markwatson.com

# 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 introduction is on general Common Lisp programming techniques. Later we will look at general example applications and artificial intelligence examples.

The Common Lisp program examples are distributed on the github repo for this book[19].

## Why Did I Write this Book?

Why the title "Loving Common Lisp"? Simple! I have been using Lisp for almost 40 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 often use Python for deep learning. I like 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 over a decade ago, 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 third edition example code so you can easily install required libraries. For the fourth and fifth editions of this book I have added more examples using neural networks and deep learning. In this new sixth edition I have added a complete application that uses CAP for the user interface.

As programmers, we all (hopefully) enjoy applying our experience and brains for 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, Python 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.

[19] https://github.com/mark-watson/loving-common-lisp

# 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 (CCL) – 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[20]
- Allegro Common Lisp - high quality, great support and higher cost. franz.com[21]
- 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

I currently (mostly) use SBCL, CCL, and LispWorks. The SBCL compiler produces very fast code and the compiler warning can be of great value in finding potential problems with your code. Like CCL because it compiles quickly so is often preferable for development.

For working through this book, I will assume that you are using SBCL or CCL. For the example in the last chapter you will need LispWorks and the free Personal edition is fine for the purposes of experimenting with the example application and the CAPI user interface library.

# 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:

[20] http://www.lispworks.com
[21] http://franz.com

```
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't legally be assigned a string value: the code in line 5 would generate a compilation error. Lisp code can assign a value to a variable and then reassign another value of a different type.

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 have used Clozure Common Lisp, SBCL, Allegro Lisp (from Franz Inc), 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 of 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 does 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 a general use 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 large 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 and when writing new 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 other languages and programming environments.

# Common Lisp Basics

The material in this chapter will serve as an introduction to Common Lisp. I have attempted to make this book a self contained resource for learning Common Lisp and to provide code examples to perform common tasks. If you already know Common Lisp and bought this book for the code examples later in this book then you can probably skip this chapter.

For working through this chapter we will be using the interactive shell, or **repl**, built into SBCL and other Common Lisp systems. For this chapter it is sufficient for you to download and install SBCL[22]. Please install SBCL right now, if you have not already done so.

## Getting Started with SBCL

When we start SBCL, we see an 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:

```
% sbcl
(running SBCL from: /Users/markw/sbcl)
This is SBCL 2.0.2, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.

* (defvar x 1.0)

X
* x

1.0
* (+ x 1)
```

[22] http://www.sbcl.org/platform-table.html

```
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 support coding in Lisp but I personally use Emacs or sometimes VScode (with Common Lisp plugins) to edit 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 2 3 4)

10
* (quit)
Bye.
```

If you get an error you can enter **help** to get options for handling an error. When I get an error and have a good idea of what caused the error then I just enter **:a** to abort out of the error).

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, Quicklisp, 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 an Emacs extension package that greatly facilitates Lisp development. SBCL is a robust Common Lisp compiler and runtime system that is often used in production.

We will cover the Quicklisp package manager and using Quicklisp to setup Slime and Emacs in a later chapter.

I will not spend much time covering the use of Emacs as a text editor in this book since you can try most of the example code snippets in the book text by copying and then pasting them into a SBCL repl and by loading the book example source files directly into a repl. If you already use Emacs then I recommend that you do set up Slime sooner rather than later and start using it for development. If you are not already an Emacs user and do not mind spending the effort to learn Emacs, then search the web first for an Emacs tutorial. That said, you will easily be able to use the example code from this book using any text editor you like with a SBCL repl. I don't use the vi or vim editors but if vi is your weapon of choice for editing text then a web search for "common lisp vi vim repl" should get you going for developing Common Lisp code with vi or vim. If you are not already an Emacs or vi user then using VSCode with a Common Lisp plugin is recommended.

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.

## 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 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[23] installed, install rlwrap with:

[23] http://mxcl.github.io/homebrew/

```
brew install rlwrap
```

If you are running Ubuntu Linux, install rlwrap using:

```
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**:

```
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:

```
-> ~ which sbcl
/Users/markw/sbcl/sbcl
-> ~ 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
* 3.14159
3.14159
* "the dog bit the cat"
"the dog bit the cat"
* (defun my-add-one (x)
(+ x 1))
MY-ADD-ONE
* (my-add-one -10)
-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):

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

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

```
* (setq x '(1 2 3))
(1 2 3)
* (typep x 'list)
T
* (typep x 'sequence)
T
* (typep x 'number)
NIL
* (typep (+ 1 2 3) 'number)
T
* (type-of 3.14159)
single-float
* (type-of "the dog ran quickly")
(simple-array character (19))
```

```
* (type-of 100193)
(integer 0 4611686018427387903)
```

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 2 3 4 5)
15
* *
15
* (setq x *)
15
* x
15
```

All Common Lisp environments set * to the value of the last expression evaluated. This example may be slightly confusing because * is also the prompt character in the SBCL repl that indicates that you can enter a new expression for evaluation. For example in line 3, the first * character is the repl prompt and the second * we type in to see that value of the previous expression that we typed into the repl.

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. We can nest **defun** special form inside each other and inside **let** expressions but this defines the nested functions globally. We use the special forms **flet** and **labels** to define functions inside a scoped environment. Functions defined inside a **labels** special form can be recursive while functions defined inside a **flet** special form cannot be recursive. Consider the following example in the file **nested.lisp** (all example files are in the **src** directory):

```
1  (flet ((add-one (x)
2           (+ x 1))
3         (add-two (x)
4           (+ x 2)))
5    (format t "redefined variables: ~A  ~A~%" (add-one 100) (add-two 100)))
6
7  (let ((a 3.14))
8    (defun test2 (x)
9      (print x))
10   (test2 a))
11
12 (test2 50)
13
14 (let ((x 1)
15       (y 2))
16   ;; define a test function nested inside a let statement:
17   (flet ((test (a b)
18            (let ((z (+ a b)))
19              ;; define a helper function nested inside a let/function/let:
20              (flet ((nested-function (a)
21                       (+ a a)))
22                (nested-function z)))))
23     ;; call nested function 'test':
24     (format t "test result is ~A~%" (test x y))))
25
26 (let ((z 10))
27   (labels ((test-recursion (a)
28              (format t "test-recursion ~A~%" (+ a z))
29              (if (> a 0)
30                  (test-recursion (- a 1)))))
31     (test-recursion 5)))
```

We define a top level **flet** special form in lines 1-5 that defines two nested functions **add-one** and **add-two** and then calls each nested function in the body of the **flet** special form. For many years I have used nested **defun** special forms inside **let** expressions for defining local functions and you will notice this use in a few later examples. However, functions defined inside **defun** special forms have global visibility so they are not hidden in the local context where they are defined. The example of a nested **defun** in lines 7-12 shows that the function **test2** has global visibility inside the current package.

Functions defined inside of a **flet** special form have access to variables defined in the outer scope containing the **flet** (also applies to **labels**). We see this in lines 14-24 where the local variables **x** and **y** defined in the **let** expression are visible inside the function **nested-function** defined inside the **flet**.

The final example in lines 26-31 shows a recursive function defined inside a **labels** special form.

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** in the sub-directory **code_snippets_for_book** using the **load** function:

```
* (load "./code_snippets_for_book/nested.lisp")
redefined variables: 101  102

3.14
50 test result is 6
test-recursion 15
test-recursion 14
test-recursion 13
test-recursion 12
test-recursion 11
test-recursion 10
T
*
```

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

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

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

Notice how indexing of arrays is zero-based; that is, indices start at zero for the first element of a sequence. Also notice that array elements can be any Lisp data type. So far, we have used the special operator **setq** to set the value of a variable. Common Lisp has a generalized version of **setq** called **setf** that can set any value in a list, array, hash table, etc. You can use **setf** instead of **setq** in all cases, but not vice-versa. Here is a simple example:

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

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

# 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:

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

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

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

# 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:

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

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

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

```
* (defvar l1 '(1 2 (3) 4 (5 6)))
L1
* (length l1)

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

You can also use the function **list** to create a new list; the arguments passed to function list are the elements of the created list:

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

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

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

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

```
* (car (cdr l1))
2
* (cadr l1)
2
*
```

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

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

```
* l1
(1 2 (3) 4 (5 6))
* (cadr l1)
2
* (caddr l1)
(3)
(car (caddr l1))
3
* (caar (last l1))
5
* (caar (cddddr l1))

5
*
```

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

```
* (last l1)
((5 6))
*
```

Common list supplies alternative functions to **car** and **cdr** that you might find more readable: **first**, **second**, **third**, **fourth**, and **rest**. Here are some examples:

```
* (defvar *x* '(1 2 3 4 5))

*X*
* (first *x*)

1
* (rest *x*)

(2 3 4 5)
* (second *x*)

2
* (third *x*)

3
* (fourth *x*)

4
```

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

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

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

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

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

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

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

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

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

# 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:

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

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

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

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

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

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

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

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

# 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

# Using Hash Tables

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

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

The second argument to function **assoc** is a list of cons cells. Function **assoc** searches for a sub-list (in the second argument) that has its **car** (i.e., first element) equal to the first argument to function **assoc**. The perhaps surprising thing about this example is that **assoc** seems to work with an integer as the first argument but not with a string. The reason for this is that by default the test for equality is done with **eql** that tests two variables to see if they refer to the same memory location or if they are identical if they are numbers. In the last call to **assoc** we used ":test #'equal" to make **assoc** use the function **equal** to test for equality.

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

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

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

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

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

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

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

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

```
* (setf (gethash 'bb h) 'aa)
AA
* (gethash 'bb h)
AA ;
T
*
```

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

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

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

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

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

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

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:

```
* (hash-table-count h2)
2
* (remhash "animal" h2)
T
* (hash-table-count h2)
1
* (clrhash h2)
#S(HASH-TABLE EQUAL)
* (hash-table-count h2)
0
*
```

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

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

# 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:

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

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

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

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

I do my professional Lisp programming using free software tools: Emacs, 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:

```
* (defun my-add-one (x) (+ x 1))

MY-ADD-ONE
* (my-add-one 10)

11
* (my-add-one 3.14159)

4.14159
* (my-add-one "cat")

debugger invoked on a SIMPLE-TYPE-ERROR: Argument X is not a NUMBER: "cat"

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

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

(SB-KERNEL:TWO-ARG-+ "cat" 1)
0] help
```

```
The debug prompt is square brackets, with number(s) indicating the current
  control stack level and, if you've entered the debugger recursively, how
  deeply recursed you are.

 ...

Getting in and out of the debugger:
  TOPLEVEL, TOP  exits debugger and returns to top level REPL
  RESTART        invokes restart numbered as shown (prompt if not given).
  ERROR          prints the error condition and restart cases.

 ...

Inspecting frames:
  BACKTRACE [n]  shows n frames going down the stack.
  LIST-LOCALS, L lists locals in current frame.
  PRINT, P       displays function call for current frame.
  SOURCE [n]     displays frame's source form with n levels of enclosing forms.

Stepping:
  START Selects the CONTINUE restart if one exists and starts
        single-stepping. Single stepping affects only code compiled with
        under high DEBUG optimization quality. See User Manual for details.
  STEP  Steps into the current form.
  NEXT  Steps over the current form.
  OUT   Stops stepping temporarily, but resumes it when the topmost frame that
        was stepped into returns.
  STOP  Stops single-stepping.

 ...

0] list-locals
SB-DEBUG::ARG-0  =  "cat"
SB-DEBUG::ARG-1  =  1

0] backtrace 2

Backtrace for: #<SB-THREAD:THREAD "main thread" RUNNING {1002AC32F3}>
0: (SB-KERNEL:TWO-ARG-+ "cat" 1)
1: (MY-ADD-ONE "cat")
0] :0

*
```

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.

The last two chapters of this book provide example applications that are configured to work with Quicklisp, which we will study in the next chapter.

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:

```
(defun non-functional-example (car)
  (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.

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.

# 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[24]. You will need to have Quicklisp installed for many of the examples later in this book so please take the time to install it now as per the instructions on the Quicklisp web site.

## Using Quicklisp to Find Packages

We will need the Common Lisp Hunchentoot library later in the Chapter on Network Programming so we will install it now using Quicklisp as an example for getting started with Quicklisp.

We already know the package name we want, but as an example of discovering packages let's start by using Quicklisp to search for all packages with "hunchentoot" in the package name:

```
1  * (ql:system-apropos "hunchentoot")
2  #<SYSTEM clack-handler-hunchentoot / clack-20131111-git / quicklisp 2013-11-11>
3  #<SYSTEM hunchentoot / hunchentoot-1.2.21 / quicklisp 2013-11-11>
4  #<SYSTEM hunchentoot-auth / hunchentoot-auth-20101107-git / quicklisp 2013-11-11>
5  #<SYSTEM hunchentoot-cgi / hunchentoot-cgi-20121125-git / quicklisp 2013-11-11>
6  #<SYSTEM hunchentoot-dev / hunchentoot-1.2.21 / quicklisp 2013-11-11>
7  #<SYSTEM hunchentoot-single-signon / hunchentoot-single-signon-20131111-git / quickl\
8  isp 2013-11-11>
9  #<SYSTEM hunchentoot-test / hunchentoot-1.2.21 / quicklisp 2013-11-11>
10 #<SYSTEM hunchentoot-vhost / hunchentoot-vhost-20110418-git / quicklisp 2013-11-11>
```

We want the base package seen in line 3 and we can install the base package as seen in the following example:

[24] http://www.quicklisp.org/

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

In line 1, I refer to the package name using a symbol :hunchentoot but using the string "hunchentoot" would have worked the same. The first time you **ql:quickload** a library you may see additional printout and it takes longer to load because the source code is downloaded from the web and cached locally in the directory **~/quicklisp/local-projects**. 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 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 most 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:

```
(ql:quickload "clsql")
(ql:quickload "clsql-postgresql")
(ql:quickload "clsql-mysql")
(ql:quickload "clsql-sqlite3")
(ql:quickload :drakma)
(ql:quickload :hunchentoot)
(ql:quickload :cl-json)
(ql:quickload "clouchdb")  ;; for CouchDB access
(ql:quickload "sqlite")
```

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. If you are unlikely to use relational databases with Common Lisp then you might skip the effort of installing Postgres and MySQL. The example in the Chapter on the Knowledge Graph Navigator uses the SQLite database for caching. You don't need any extra dependencies for the **sqlite** package.

## 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:

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

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

```
[package quicklisp-slime-helper]
slime-helper.el installed in "/Users/markw/quicklisp/slime-helper.el"

To use, add this to your ~/.emacs:

  (load (expand-file-name "~/quicklisp/slime-helper.el"))
  ;; Replace "sbcl" with the path to your implementation
  (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:

```
(setq inferior-lisp-program "/Users/markw/sbcl/sbcl")
```

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

In later chapters we will write libraries and applications as Quicklisp projects so that you will be able to load your own libraries, making it easier to write small libraries that you can compose into larger applications.

# 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:

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

Then in this example, if you see the variable ***lexical-hash-table*** inside a function definition, 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:

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

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

```
* (defun test (x)
       (let ((y (list x x)))
         y))
TEST
* (test "the dog bit the cat")
("the dog bit the cat" "the dog bit the cat")
*
```

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

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

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

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

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

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

## 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:

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

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

Consider the following repl listing where we will look at a primary difference between quoting a symbol using ' and with **#'**:

```
$ ccl
Clozure Common Lisp Version 1.12  DarwinX8664
? 'barfoo531
BARFOO531
? (apropos "barfoo")
BARFOO531
? #'bar987
> Error: Undefined function: BAR987
```

On line three we create a new symbol **BARFOO531** that is interned as you can see from looking at all interned symbols containing the string "barfoo". Line 7 throws an error because **#'** does not intern a new symbol.

Here is the example file **src/lambda1.lisp**:

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

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

```
* (test)
2

*
```

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

```
* (defvar f1 #'(lambda (x) (+ x 1)))

F1
* (funcall f1 100)

101
* (funcall #'print 100)

100
100
```

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

## 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:

```
;; a simple loop using recursion

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

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

```
entering recursion1(5)
NIL
* (recursion1 20)
entering recursion1(20)
NIL
*
```

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:

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

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

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

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

## Using the Function eval

In Lisp languages we often say that code is data. The function **eval** can be used to execute code that is stored as Lisp data. Let's look at an example:

```
$ ccl
Clozure Common Lisp Version 1.12  DarwinX8664
? '(+ 1 2.2)
(+ 1 2.2)
? (eval '(+ 1 2.2))
3.2
? (eval '(defun foo2 (x) (+ x x)))
FOO2
? (foo2 4)
8
```

I leave it up to you, dear reader, how often you are motivated to use **eval**. In forty years of using Lisp languages my principle use of **eval** has been in modifying the standard version of the Ops5 programming language for production systems[25] to support things like multiple data worlds and new actions to spawn off new data worlds and to remove them. Ops5 works by finding common expressions in a set of production rules (also referred to as "expert systems") and factoring them into a network (a Rete network if you want to look it up) with common expressions in rules stored in just a single place. **eval** is used a lot in Ops5 and I used it for my extensions to Ops5.

[25] https://github.com/sharplispers/ops5

# 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.

The ability to add functionality and syntax to the Common Lisp language, to in effect extend the language as needed, is truly a super power of languages like Common Lisp and Scheme.

## 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:

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

Notice that the example macro **double-list** itself uses the macro **dolist**. It is common to nest macros in the same way functions can be nested.

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

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

## Using the Splicing Operator

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

```
;; another macro example that uses ,@:

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

;; use the macro:

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

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

## Using macroexpand-1

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

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

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

We have only "scratched the surface" looking at macros; the interested reader is encouraged to search the web using, for example, "Common Lisp macros." There are two books in particular that I recommend that take a deep dive into Common Lisp macros: Paul Graham's "On Lisp" and Doug Hoyte's "Let Over Lambda." Both are deep books and will change the way you experience software development. A good plan of study is spending a year absorbing "On Lisp" before tackling "Let Over Lambda."

# 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:

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

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

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

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

## dotimes

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

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

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

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

0
1
2
"all-done-with-test-dotimes-loop"
*
```

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

# do

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

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

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

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

;; example do macro use

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

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

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

```
* (load "do1.lisp")
;; Loading file do1.lisp ...
0
1
2
3
;; Loading of file do1.lisp is finished.
T
*
```

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

## Using the loop Special Form to Iterate Over Vectors or Arrays

We previousely used **dolist** to iterate over elements in lists. For efficiency we will often use vectors (one dimensional arrays) and we can use **loop** to similarly handle vectors:

```
(loop for td across testdata
   do
     (print td))))
```

where **testdata** is a one dimensional array (a vector) and inside the **do** block the local variable **td** is assigned to each element in the vector.

# Common Lisp Package System

In later chapters we will see two complete applications that are defined as Quicklisp projects: the chapter on the Knowledge Graph Creator and the chapter on the Knowledge Graph Navigator. Another example for setting up a Quicklib project can be seen in the chapter Plotting Data.

While these later chapters provide practical examples for bundling up your own projects in packages, the material here will give you general background information that you should know.

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

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

As we will use in the following example, the package **:cl** is an alias for **:common-lisp-user**.

We will define a new package :my-new-package and two functions **foo1** and **foo2** inside the package. Externally to this package, assuming that it is loaded, we can access **foo2** using **my-new-package:foo2**. **foo1** is not exported so it cannot be accessed this way. However, 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 so we can use **my-new-package::foo1**. Using :: allows us access to symbols not explicitly exported.

When I leave package **:my-new-package** in line 22 and return to package **:cl**, and try to access **my-new-package:foo1** notice that an error is thrown.

On line 3 we define the alias **:p1** for the package **:my-new-package** and we use this alias in line 44. The main point of the following example is that we define two functions in a package but only export one of these functions. By default the other function is not visible outside of the new package.

```
1  * (defpackage "MY-NEW-PACKAGE"
2      (:use :cl)
3      (:nicknames "P1")
4      (:export :FOO2))
5
6  #<PACKAGE "MY-NEW-PACKAGE">
7  * (in-package my-new-package)
8
9  #<PACKAGE "MY-NEW-PACKAGE">
10 * (defun foo1 () "foo1")
```

```
11
12 FOO1
13 * (defun foo2 () "foo2")
14
15 FOO2
16 * (foo1)
17
18 "foo1"
19 * (foo2)
20
21 "foo2"
22 * (in-package :cl)
23
24 #<PACKAGE "COMMON-LISP">
25 * (my-new-package:foo2)
26
27 "foo2"
28 * (my-new-package:foo1)
29
30 debugger invoked on a SB-INT:SIMPLE-READER-PACKAGE-ERROR in thread
31 #<THREAD "main thread" RUNNING {1001F1ECE3}>:
32   The symbol "FOO1" is not external in the MY-NEW-PACKAGE package.
33
34     Stream: #<SYNONYM-STREAM :SYMBOL SB-SYS:*STDIN* {100001C343}>
35
36 Type HELP for debugger help, or (SB-EXT:EXIT) to exit from SBCL.
37
38 restarts (invokable by number or by possibly-abbreviated name):
39   0: [CONTINUE] Use symbol anyway.
40   1: [ABORT   ] Exit debugger, returning to top level.
41
42 * 1
43
44 * (p1:foo2)
45
46 "foo2"
```

Since we specified a nickname in the **defpackage** expression, Common Lisp allows the use of the nickname (in this case **P1**) in calling function **foo2** that is exported from 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** on line 42.

What about the error on line 28 where **my-new-package:foo1** is undefined because the function

**foo1** is not exported (see line 4)? It turns out that you can easily use symbols not exported from a package by using :: instead of a single :. Here, this would be defined: **(my-new-package::foo1)**.

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.

For small packages I sometimes put a **defpackage** expression at the top of the file immediately followed by an in-package expression to switch to the new package. In the general case, please properly use separate **project** and **asdf** files as I do in the later chapters Knowledge Graph Creator and Knowledge Graph Navigator.

# 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, files, 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. In the later chapter Knowledge Graph Navigator that supports a user interface, we will again use output streams bound to different scrolling output areas of the application window to write color-hilighted text. The stream formalism is general purpose, covering many common I/O use cases.

## The Lisp read and read-line Functions

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

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

```
        read with-open-file
```

In the function **read-test-1**, we use the macro with-open-file to read from a file. To write to a file (which we will do later), we 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:

```
(defun read-test-1 ()
  "read a maximum of 1000 expressions from the file 'test.dat'"
  (with-open-file
   (input-stream "test.dat" :direction :input)
   (dotimes (i 1000)
     (let ((x (read input-stream nil nil)))
       (if (null x) (return)) ;; break out of the 'dotimes' loop
       (format t "next expression in file: ~S~%" x)))))
```

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

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

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

In this last example, we passed the file name as a string to the macro **with-open-file**. This is not 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:

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

Here, we are specifying that we want to use 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

```
(defun readline-test ()
  "read a maximum of 1000 expressions from the file 'test.dat'"
  (with-open-file
   (input-stream "test.dat" :direction :input)
   (dotimes (i 1000)
     (let ((x (read-line input-stream nil nil)))
       (if (null x) (return)) ;; break out of the 'dotimes' loop
       (format t "next line in file: ~S~%" x)))))
```

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

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

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

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

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

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

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

## 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:

```
* (print "testing")

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

"testing"
"testing"
*
```

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

```
* (print "testing")

"testing"
"testing"
* (princ "testing")
testing
"testing"
*
```

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

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

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

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

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

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

```
% cat test1.dat

"the cat ran down the road" 1 + 2 is: 3
Stoking!!
```

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

# Plotting Data

We will use Zach Beane's vecto library[26] for plotting data with the results written to files. Ideally we would like to have interactive plotting capability but for the purposes of this book I need to support the combinations of all Common Lisp implementations on multiple operating systems. Interactive plotting libraries are usually implementation and OS dependent. We will use the **plotlib** example we develop in the later chapter Backpropagation Neural Networks.

## Implementing the Library

The examples here are all contained in the directory **src/plotlib** and is packaged as a Quicklisp loadable library. This library will be used in later chapters.

When I work on my macOS laptop, I leave the output graphics file open in the Preview App and whenever I rerun a program producing graphics in the REPL, making the preview App window active refreshes the graphics display.


**PNG file generated by running plotlib test**

The following listing shows the file *plotlib.lisp* that is a simple wrapper for the *vecto* Common Lisp plotting library. Please note that I only implemented wrappers for *vecto* functionality that I need for later examples in this book, so the following code is not particularly general but should be easy enough for you to extend for the specific needs of your projects.

[26] http://xach.com/lisp/vecto/

```
;; Misc. plotting examples using the vecto library

(ql:quickload :vecto) ;; Zach Beane's plotting library
(defpackage #:plotlib
  (:use #:cl #:vecto))

(in-package #:plotlib)

;; the coordinate (0,0) is the lower left corner of the plotting area.
;; Increasing the y coordinate is "up page" and increasing x is "to the right"

;; fills a rectangle with a gray-scale value
(defun plot-fill-rect (x y width height gscale) ; 0 < gscale < 1
  (set-rgb-fill gscale gscale gscale)
  (rectangle x y width height)
  (fill-path))

;; plots a frame rectangle
(defun plot-frame-rect (x y width height)
  (set-line-width 1)
  (set-rgb-fill 1 1 1)
  (rectangle x y width height)
  (stroke))

(defun plot-line(x1 y1 x2 y2)
  (set-line-width 1)
  (set-rgb-fill 0 0 0)
  (move-to x1 y1)
  (line-to x2 y2)
  (stroke))

(defun plot-string(x y str)
  (let ((font (get-font "OpenSans-Regular.ttf")))
    (set-font font 15)
    (set-rgb-fill 0 0 0)
    (draw-string x y str)))

(defun plot-string-bold(x y str)
  (let ((font (get-font "OpenSans-Bold.ttf")))
    (set-font font 15)
    (set-rgb-fill 0 0 0)
    (draw-string x y str)))

```

```
44
45 (defun test-plotlib (file)
46   (with-canvas (:width 90 :height 90)
47     (plot-fill-rect 5 10 15 30 0.2) ; black
48     (plot-fill-rect 25 30 30 7 0.7) ; medium gray
49     (plot-frame-rect 10 50 30 7)
50     (plot-line 90 5 10 5)
51     (plot-string 10 65 "test 1 2 3")
52     (plot-string-bold 10 78 "Hello")
53     (save-png file)))
54
55 ;;(test-plotlib "test-plotlib.png")
```

This plot library is used in later examples in the chapters on search, backpropagation neural networks and Hopfield neural networks. I prefer using implementation and operating specific plotting libraires for generating interactive plots, but the advantage of writing plot data to a file using the **vecto** library is that the code is portable across operating systems and Common Lisp implementations.

## Packaging as a Quicklisp Project

The two files **src/plotlib/plotlib.asd src/plotlib/package.lisp** configure the library. The file **package.lisp** defines the required library **vecto** and lists the functions that are publicly exported from the library:

```
(defpackage #:plotlib
  (:use #:cl #:vecto)
  (:export save-png plot-fill-rect plot-frame-rect
     plot-size-rect plot-line plot-string plot-string-bold
     pen-width))
```

To run the test function provided with this library you load the library and preface exported function names with the package name **plotlib**: as in this example:

```
(ql:quickload "plotlib")
(plotlib::test-plotlib "test-plotlib.png")
```

In addition to a **package.lisp** file we also use a file with the extension **.asd**

```
(asdf:defsystem #:plotlib
  :description "Describe plotlib here"
  :author "mark.watson@gmail.com"
  :license "Apache 2"
  :depends-on (#:vecto)
  :components ((:file "package")
               (:file "plotlib")))
```

If you have specified a dependency that is not already downloaded to your computer, Quicklisp will install the dependency for you.

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

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

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

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

There are four generic functions used in the function test:

- set-header - required to initialize class and also defines the page title
- add-element - used to insert a string that defines any type of HTML element
- 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/loving_snippets/HTMLstream.lisp**. We start defining a new class using the macro **defclass** that takes the following arguments:

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

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

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

Here, the class name is **HTMLstream**, the list of super classes is an empty list (), the list of slot specifications contains only one slot specification for the slot named **out** and there is only one class specification: a documentation string. Slots are like instance variables in languages like Java and Smalltalk. 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:

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

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

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

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

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

The method **add-table** 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:

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

```
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**:

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

## Using Defstruct or CLOS

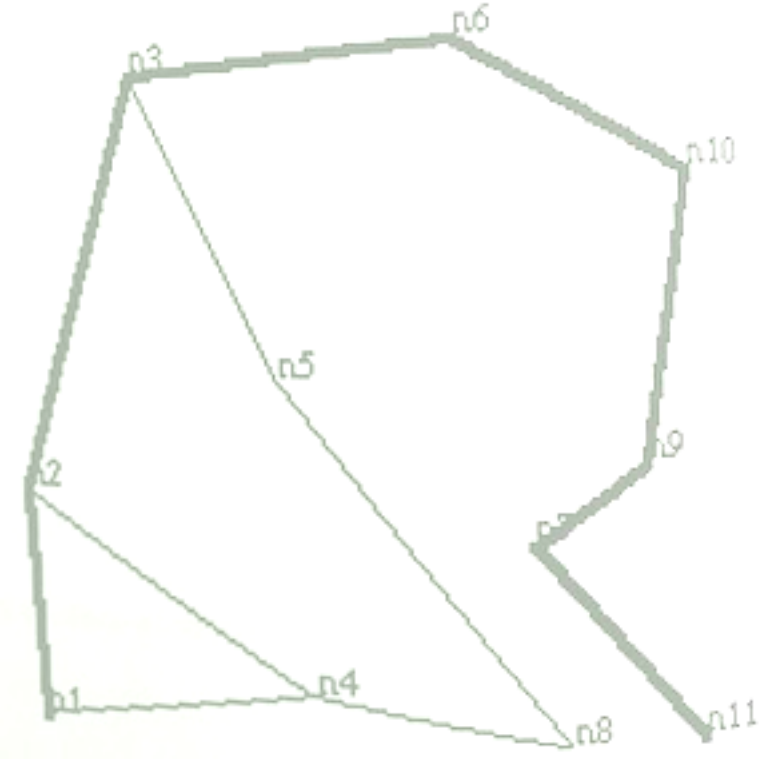
You might notice from my own code that I use Common Lisp **defstruct** macros to define data structures more often than I use CLOS. The **defclass** macro used to create CLOS classes are much more flexible but for simple data structures I find that using **defstruct** is much more concise. In the simplest case, a **defstruct** can just be a name of the new type followed by slot names. For each slot like **my-slot-1** accessor functions are generated automatically. Here is a simple example:

```
$ ccl
Clozure Common Lisp Version 1.12  DarwinX8664
? (defstruct struct1 s1 s2)
STRUCT1
? (make-struct1 :s1 1 :s2 2)
#S(STRUCT1 :S1 1 :S2 2)
? (struct1-s1 (make-struct1 :s1 1 :s2 2))
1
```

We defined a struct **struct1** on line3 with two slots names **s1** and **s2**, show the use of the automatically generated constructor **make-struct1** on line 5, and one of the two automatically generated accessor functions **struct1-s1** on line 7. The names of accessor functions are formed with the structure name and the slot name.

# Heuristically Guided Search

We represent search space as a graph: nodes and links between the nodes. The following figure shows the simple graph that we use as an example, finding a route from node **n1** to node **n11**:




**Plot of best route using the plotlib utilities**

The following example code uses a heuristic for determining which node to try first from any specific location: move to the node that is closest spatially to the goal node. We see that this heuristic will not always work to produce the most efficient search but we will still get to the goal node. As an example in which the heuristic does not work, consider when we start at node **n1** in the lower left corner of the figure. The search algorithm can add nodes **n2** and **n4** to the nodes to search list and will search using node **n4** first since **n4** is closer to the goal node **n11** than node **n2**. In this case, the search will eventually need to back up trying the path **n1** to **n2**. Despite this example of the heuristic not working to decrease search time, in general, for large search spaces (i.e., graphs with many nodes and edges) it can dramatically decrease search time.

The main function **A*search** starting in line 5 extends to line 151 because all search utility functions are nested (lexically scoped) inside the mani function. The actual code for the main function **A*search** is in lines 150 and 151.

The data representing nodes in this implementation is globally scoped (see the definitions on lines 155-165 in the "throw away test code" near the bottom of the file) and we set the property **path-list** to store the nodes directy connected to each node (set in function **init-path-list** in lines 36-52). I originally wrote this code in 1990 which explains it non-functional style using globally scoped node variables.

```
1  ;; Perform a heuristic A* search between the start and goal nodes:
2  ;;
3  ;; Copyright 1990, 2017 by Mark Watson
4
5  (defun A*search (nodes paths start goal &aux possible-paths best)
6
7    (defun Y-coord (x) (truncate (cadr x)))
8    (defun X-coord (x) (truncate (car x)))
9
10   (defun dist-between-points (point1 point2)
11     (let ((x-dif (- (X-coord point2) (X-coord point1)))
12           (y-dif (- (Y-coord point2) (Y-coord point1))))
13       (sqrt (+ (* x-dif x-dif)  (* y-dif y-dif)))))
14
15   (setq possible-paths
16         (list
17          (list
18           (dist-between-points
19            (eval start)
20            (eval goal))
21           0
22           (list start))))
23
24   (defun init-network ()
25     (setq paths (init-lengths paths))
26     (init-path-list nodes paths))
27
28   (defun init-lengths (pathlist)
29     (let (new-path-list pathlength path-with-length)
30       (dolist (path pathlist)
31         (setq pathlength (slow-path-length path))
32         (setq path-with-length (append path (list pathlength)))
33         (setq new-path-list (cons path-with-length new-path-list)))
34       new-path-list))
35
36   (defun init-path-list (nodes paths)
37     (dolist (node nodes)
```

```
      (setf
       (get node 'path-list)
       ;; let returns all paths connected to node:
       (let (path-list)
         (dolist (path paths)
           (if (equal node (start-node-name path))
               (setq path-list
                     (cons (list (end-node-name path)
                                 (path-length path))
                           path-list))
               (if (equal node (end-node-name path))
                   (setq path-list (cons (list (start-node-name path)
                                               (path-length path))
                                         path-list)))))
         path-list ))))

(defun slow-path-length (path)
  (dist-between-points (start-node path) (end-node path)))

(defun path-length (x) (caddr x))

(defun start-node (path) (eval (car path)))
(defun end-node (path) (eval (cadr path)))
(defun start-node-name (x) (car x))
(defun end-node-name (x) (cadr x))
(defun first-on-path (x) (caddr x))
(defun goal-node (x) (car x))
(defun distance-to-that-node (x) (cadr x))

(defun enumerate-children (node goal)
  (let* ((start-to-lead-node-dist (cadr node)) ;; distance already calculated
         (path (caddr node))
         (lead-node (car path)))
    (if (get-stored-path lead-node goal)
        (consider-best-path lead-node goal path start-to-lead-node-dist)
        (consider-all-nodes lead-node goal path start-to-lead-node-dist))))

(defun consider-best-path (lead-node goal path distance-to-here)
  (let* (
         (first-node (get-first-node-in-path lead-node goal))
         (dist-to-first (+ distance-to-here
                           (get-stored-dist lead-node first-node)))
         (total-estimate (+ distance-to-here
```

```
                              (get-stored-dist lead-node goal)))
           (new-path (cons first-node path)))
      (list (list total-estimate dist-to-first new-path))))

(defun get-stored-path (start goal)
  (if (equal start goal)
      (list start 0)
      (assoc goal (get start 'path-list))))

(defun node-not-in-path (node path)
  (if (null path)
      t
      (if (equal node (car path))
          nil
          (node-not-in-path node (cdr path)))))

(defun consider-all-nodes (lead-node goal path start-to-lead-node-dist)
  (let (dist-to-first total-estimate new-path new-nodes)
    (dolist (node (collect-linked-nodes lead-node))
      (if (node-not-in-path node path)
          (let ()
            (setq dist-to-first (+ start-to-lead-node-dist
                                   (get-stored-dist lead-node node)))
            (setq total-estimate (+ dist-to-first
                                    (dist-between-points
                                     (eval node)
                                     (eval  goal))))
            (setq new-path (cons node path))
            (setq new-nodes (cons (list total-estimate
                                        dist-to-first
                                        new-path)
                                  new-nodes)))))
    new-nodes))

(defun collect-linked-nodes (node)
  (let (links)
    (dolist (link (get node 'path-list))
      (if (null (first-on-path link))
          (setq links (cons (goal-node link) links))))
    links))

(defun get-stored-dist (node1 node2)
  (distance-to-that-node (get-stored-path node1 node2)))
```

```
124
125   (defun collect-ascending-search-list-order (a l)
126     (if (null l)
127         (list a)
128         (if (< (car a) (caar l))
129             (cons a l)
130             (cons (car l) (Collect-ascending-search-list-order a (cdr l))))))
131
132   (defun get-first-node-in-path (start goal)
133     (let (first-node)
134       (setq first-node  (first-on-path (get-stored-path start goal)))
135       (if first-node first-node goal)))
136
137   (defun a*-helper ()
138     (if possible-paths
139         (let ()
140           (setq best (car possible-paths))
141           (setq possible-paths (cdr possible-paths))
142           (if (equal (first (caddr best)) goal)
143               best
144               (let ()
145                 (dolist (child (enumerate-children best goal))
146                   (setq possible-paths
147                         (collect-ascending-search-list-order
148                          child possible-paths)))
149                 (a*-helper))))))
150   (init-network)
151   (reverse (caddr (a*-helper))))
152
153 ;;      Throw away test code:
154
155 (defvar n1 '(30 201))
156 (defvar  n2 '(25 140))
157 (defvar  n3 '(55 30))
158 (defvar  n4 '(105 190))
159 (defvar  n5 '(95 110))
160 (defvar  n6 '(140 22))
161 (defvar  n7 '(160 150))
162 (defvar  n8 '(170 202))
163 (defvar  n9 '(189 130))
164 (defvar  n10 '(200 55))
165 (defvar  n11 '(205 201))
166
```

```
(print (A*search
        '(n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11) ;; nodes
        '((n1 n2) (n2 n3) (n3 n5) (n3 n6) (n6 n10) ;; paths
          (n9 n10) (n7 n9) (n1 n4) (n4 n2) (n5 n8)
          (n8 n4) (n7 n11))
        'n1 'n11)) ;; starting and goal nodes
```

The following example in the repl shows the calculation of the path that we saw in the figure of the graph search space.

```
$ sbcl
* (load "astar_search.lisp")

(N1 N2 N3 N6 N10 N9 N7 N11)
T
*
```

There are many types of search: breadth first as we used here, depth first, with heuristics to optimize search dependent on the type of search space.

# 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 be learning how to use Edi Weitz's Drakma HTTP client library[27]. In order to have a complete client server example we will also look briefly at Edi Weitz's Hunchentoot web server[28] 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[29] 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**:

[27] http://weitz.de/drakma/
[28] http://weitz.de/hunchentoot/
[29] http://weitz.de/drakma/

```
* (ql:quickload :drakma)
* (multiple-value-setq
    (data http-response-code headers)
    (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**:

```
* data

"<!DOCTYPE html>
<html>
  <head>
        <title>Mark Watson: Consultant and Author</title>
```

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

```
* http-response-code

200
```

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

```
* headers

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

We will use Drakma later in this book for several examples. In the next section we will write a web app using Hunchentoot and test it with a Drakma client.

# An introduction to Hunchentoot

Edi Weitz's Hunchentoot project[30] 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:

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

I will use only easy handler framework[31] in the Hunchentoot examples in this section. I leave it to you to read the documentation on using custom acceptors[32] 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):

```
(ql:quickload :hunchentoot)
(ql:quickload :cl-who)

(in-package :cl-user)
(defpackage hdemo
  (:use :cl
        :cl-who
        :hunchentoot))
(in-package :hdemo)

(defvar *h* (make-instance 'easy-acceptor :port 3000))

;; define a handler with the arbitrary name my-greetings:

(define-easy-handler (my-greetings :uri "/hello") (name)
  (setf (hunchentoot:content-type*) "text/html")
  (with-html-output-to-string (*standard-output* nil :prologue t)
    (:html
     (:head (:title "hunchentoot test"))
     (:body
      (:h1 "hunchentoot form demo")
      (:form
       :method :post
       (:input :type :text
               :name "name"
```

[30] http://weitz.de/hunchentoot/
[31] http://weitz.de/hunchentoot/#easy-handlers
[32] http://weitz.de/hunchentoot/#acceptors

```
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 filed 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:

```
* (drakma:http-request "http://127.0.0.1:3000/hello?name=Mark")

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

We will use both Drackma and Hunchentoot in the next section.

# 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:

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

Y
* y

(((CAT . "the cat ran") (DOG . 101)) 1 2 3 4 5)
* (json:encode-json-to-string y)
"[{\"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 3000))
5
6  ;; define a handler with the name animal:
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 ue the Drakma library in another repl to test it (not all output is shown in the next listing):

```
* (ql:quickload :drakma)
* (drakma:http-request "http://127.0.0.1:3000/animal?name=dog")

"[{\"average_weight\":40,
   \"friendly\":true},
   \"A dog is a loyal creature, much valued by humans.\"]"
200
* (drakma:http-request "http://127.0.0.1:3000/animal?name=cat")

"[{\"average_weight\":10,
   \"friendly\":null},
   \"A cat can live indoors or outdoors.\"]"
200
```

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

```
* (ql:quickload :cl-json)
To load "cl-json":
  Load 1 ASDF system:
    cl-json
; Loading "cl-json"
.
(:CL-JSON)
* (cl-json:decode-json-from-string
    (drakma:http-request "http://127.0.0.1:3000/animal?name=dog"))

(((:AVERAGE--WEIGHT . 40) (:FRIENDLY . T))
 "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.

```
(defparameter *animal-hash* (make-hash-table))

;; handle HTTP POST requests:
(hunchentoot:define-easy-handler (some-handler :uri "/add") (json-data)
  (setf (hunchentoot:content-type*) "text/plain")
  (let* ((data-string (hunchentoot:raw-post-data :force-text t))
         (data (cl-json:decode-json-from-string json-data))
         ;; assume that the name of the animal is a hashed value:
         (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.

## Network Programming Wrap Up

You have learned the basics for writing web services and writing clients to use web services. Later, we will use web services written in Python by writing Common Lisp clients: we will wrap retrained deep learning models and access them from Common Lisp.

# Using the Microsoft Bing Search APIs

I have used the Bing search APIs for many years. Microsoft Bing supports several commercial search engine services, including my favorite search engine Duck Duck Go. Bing is now part of the Azure infrastructure that is branded as "Cognitive Services." You should find the example code for this chapter relatively easy to extend to other Azure Cognitive Services that you might need to use.

You will need to register with Microsoft's Azure search service to use the material in this chapter. It is likely that you view search as a manual human-centered activity. I hope to expand your thinking to considering applications that automate search, finding information on the web, and automatically organizing information.

While the example code uses only the search APIs, with some modification it can be extended to work with all REST APIs provided by Azure Cognitive Services[33] that include: analyzing text to get user intent, general language understanding, detecting key phrases and entity names, translate between languages, converting between speech and text, and various computer vision services. These services are generally free or very low cost for a few thousand API calls a month, with increased cost for production deployments. Microsoft spends about $1 billion a year in research and development for Azure Cognitive Services.

## Getting an Access Key for Microsoft Bing Search APIs

You will need to set up an Azure account if you don't already have one. I use the Bing search APIs fairly often for research but I have never spent more than about a dollar a month and usually I get no bill at all. For personal use it is a very inexpensive service.

You start by going to the web page https://azure.microsoft.com/en-us/try/cognitive-services/[34] and sign up for an access key. The Search APIs sign up is currently in the fourth tab in this web form. When you navigate to the Search APIs tab, select the option Bing Search APIs v7. You will get an API key that you need to store in an environment variable that you will soon need:

```
export BING_SEARCH_V7_SUBSCRIPTION_KEY=1e97834341d2291191c772b7371ad5b7
```

That is not my real subscription key!

You also set the Bing search API as an environment variable:

[33] https://azure.microsoft.com/en-us/services/cognitive-services/
[34] https://azure.microsoft.com/en-us/try/cognitive-services/

```
export BING_SEARCH_V7_ENDPOINT=https://api.cognitive.microsoft.com/bing/v7.0/search
```

## Example Search Script

Instead of using a pure Common Lisp HTTP client library I often prefer using the **curl** command run in a separate process. The **curl** utility handles all possible authentication modes, handles headers, response data in several formats, etc. We capture the output from **curl** in a string that in turn gets processed by a JSON library.

It takes very little Common Lisp code to access the Bing search APIs. The function **websearch** makes a generic web search query. The function **get-wikidata-uri** uses the **websearch** function by adding "site:wikidata.org" to the query and returning only the WikiData URI for the original search term. We will later see several examples. I will list the entire library with comments to follow:

```
(in-package #:bing)

(defun get-wikidata-uri (query)
  (let ((sr (websearch (concatenate 'string "site:wikidata.org " query))))
    (cadar sr)))

(defun websearch (query)
  (let* ((key (uiop:getenv "BING_SEARCH_V7_SUBSCRIPTION_KEY"))
         (endpoint (uiop:getenv "BING_SEARCH_V7_ENDPOINT"))
         (command
          (concatenate
           'string
           "curl -v -X GET \""  endpoint "?q="
           (drakma:url-encode query :utf-8)
           "&mkt=en-US&limit=4\""
           " -H \"Ocp-Apim-Subscription-Key: " key "\""))
         (response
          (uiop:run-program command :output :string)))
    (with-input-from-string
        (s response)
      (let* ((json-as-list (json:decode-json s))
             (values (cdadr (cddr (nth 2 json-as-list)))))
        (mapcar #'(lambda (x)
                    (let ((name (assoc :name x))
                          (display-uri (assoc :display-url x))
                          (snippet (assoc :snippet x)))
                      (list (cdr name) (cdr display-uri) (cdr snippet))))
                values)))))
```

We get the Bing access key and the search API endpoint in lines 8-9. Lines 10-16 create a complete call to the **curl* command line utility**. **We spawn a process to run **curl** and capture the string output in the variable **response** in lines 17-18. You might want to add a few print statements to see typical values for the variables **command** and **response**. The response data is JSON data encoded in a string, with straightforward code in lines 19-28 to parse out the values we want.

The following repl listing shows this library in use:

```
$ sbcl
This is SBCL 2.0.2, an implementation of ANSI Common Lisp.
* (ql:quickload "bing")
To load "bing":
  Load 1 ASDF system:
    bing
; Loading "bing"
..............
("bing")
* (bing:get-wikidata-uri "Sedona Arizona")
"https://www.wikidata.org/wiki/Q80041"
* (bing:websearch "Berlin")
(("Berlin - Wikipedia" "https://en.wikipedia.org/wiki/Berlin"
  "Berlin (/ b□□r□l□n /; German: [b□□□li□n] (listen)) is the capital and largest cit\
y of Germany by both area and population. Its 3,769,495 (2019) inhabitants make it t\
he most populous city proper of the European Union. The city is one of Germany's 16 \
federal states.")
 ("THE 15 BEST Things to Do in Berlin - 2020 (with Photos ..."
  "https://www.tripadvisor.com/Attractions-g187323-Activities-Berlin.html"
  "Book your tickets online for the top things to do in Berlin, Germany on Tripadvis\
or: See 571,599 traveler reviews and photos of Berlin tourist attractions. Find what\
 to do today, this weekend, or in August. We have reviews of the best places to see \
in Berlin. Visit top-rated & must-see attractions.")
 ("Berlin - Official Website of the City of Berlin, Capital ..."
  "https://www.berlin.de/en"
  "Official Website of Berlin: Information about the Administration, Events, Culture\
, Tourism, Hotels and Hotel Booking, Entertainment, Tickets, Public Transport, Polit\
ical System, Local Authorities and Business in Berlin.")
 ("Berlin | History, Map, Population, Attractions, & Facts ..."
  "https://www.britannica.com/place/Berlin"
  "Berlin is situated about 112 miles (180 km) south of the Baltic Sea, 118 miles (1\
90 km) north of the Czech-German border, 110 miles (177 km) east of the former inner\
-German border, and 55 miles (89 km) west of Poland. It lies in the wide glacial val\
ley of the Spree River, which runs through the centre of the city.")
 ("Berlin travel | Germany - Lonely Planet"
```

```
  "https://www.lonelyplanet.com/germany/berlin"
  "Welcome to Berlin Berlin's combo of glamour and grit is bound to mesmerise all th\
ose keen to explore its vibrant culture, cutting-edge architecture, fabulous food, i\
ntense parties and tangible history.")
 ("Berlin 2020: Best of Berlin, Germany Tourism - Tripadvisor"
  "https://www.tripadvisor.com/Tourism-g187323"
  "Berlin is an edgy city, from its fashion to its architecture to its charged polit\
ical history. The Berlin Wall is a sobering reminder of the hyper-charged postwar at\
mosphere, and yet the graffiti art that now covers its remnants has become symbolic \
of social progress.")
 ("Berlin 2020: Best of Berlin, OH Tourism - Tripadvisor"
  "https://www.tripadvisor.com/Tourism-g50087-Berlin_Ohio-Vacations.html"
  "Berlin Tourism: Tripadvisor has 11,137 reviews of Berlin Hotels, Attractions, and\
 Restaurants making it your best Berlin resource.")
 ("Berlin (band) - Wikipedia" "https://en.wikipedia.org/wiki/Berlin_(band)"
  "Berlin is the alias for vocalist Terri Nunn, as well as the American new wave ban\
d she fronts, having been originally formed in Orange County, California. The band g\
ained mainstream-commercial success with singles including \" Sex (I'm A...) \", \" \
No More Words \" and the chart-topping \" Take My Breath Away \" from the 1986 film \
Top Gun.")
 ("Berlin's official travel website - visitBerlin.de"
  "https://www.visitberlin.de/en"
  "Berlin's way to a metropolis 100 Years of Greater Berlin In 1920, modern Berlin w\
as born at one fell swoop. 8 cities, 59 rural communities and 27 manor districts uni\
te to form \"Greater Berlin\""))
*
```

I have been using the Bing search APIs for many years. They are a standard part of my application building toolkit.

# Wrap-up

You can check out the wide range of Congitive Services[35] on the Azure site. Available APIs include: language detection, speech recognition, vision libraries for object recognition, web search, and anomaly detection in data.

In addition to using automated web scraping to get data for my personal research, I often use automated web search. I find the Microsoft's Azure Bing search APIs are the most convenient to use and I like paying for services that I use.

[35] https://azure.microsoft.com/en-us/try/cognitive-services/

# 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[36] 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.

For some database applications when I know that I will always use the embedded SQLite database (i.e., that I will never want to switch to Postgres of another database) I will just use the **sqlite** library as I do in chapter Knowledge Graph Navigator.

If you have not installed CLSQL yet, then please install it now:

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

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

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

The directory **src/clsql_examples** contains the standalone example files for this chapter.

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"):

[36] http://clsql.b9.com/

```
-> ~ psql
psql (9.1.4)
Type "help" for help.
markw=# create database news;
CREATE DATABASE
```

We will use three example programs that you can find in the **src/clsql_examples** 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 to read from the table “articles”

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

```
(ql:quickload :clsql)
(ql:quickload :clsql-postgresql)

;; Postgres connection specification:
;;    (host db user password &optional port options tty).
;; The first argument to **clsql:connect** is a connection
;; specification list:

(clsql:connect '("localhost" "news" "markw" nil)
               :database-type :postgresql)

(clsql:def-view-class articles ()
  ((id
    :db-kind :key
    :db-constraints :not-null
    :type integer
    :initarg :id)
   (uri
    :accessor uri
    :type (string 60)
    :initarg :uri)
   (title
    :accessor title
    :type (string 90)
    :initarg :title)
   (text
    :accessor text
```

```
    :type (string 500)
    :nulls-ok t
    :initarg :text)))

(defun create-articles-table ()
  (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:

```
->  src git:(master) sbcl
(running SBCL from: /Users/markw/sbcl)
* (load "clsql_create_news_schema.lisp")
* (create-articles-table)
NOTICE:  CREATE TABLE / PRIMARY KEY will create implicit index
         "article_pk" for table "articles"
T
*
```

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

```
(ql:quickload :clsql)
(ql:quickload :clsql-postgresql)

;; Open connection to database and create CLOS class and database view
;; for table 'articles':
(load "clsql_create_news_schema.lisp")

(defvar *a1*
  (make-instance
    'article
    :uri "http://test.com"
    :title "Trout Season is Open on Oak Creek"
    :text "State Fish and Game announced the opening of trout season"))

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

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

```
(ql:quickload :clsql)
(ql:quickload :clsql-postgresql)

;; Open connection to database and create CLOS class and database view
;; for table 'articles':
(load "clsql_create_news_schema.lisp")

(defun pp-article (article)
  (format t
    "~%URI: ~S ~%Title: ~S ~%Text: ~S ~%"
    (slot-value article 'uri)
    (slot-value article 'title)
    (slot-value article 'text)))

(dolist (a (clsql:select 'article))
  (pp-article (car a)))
```

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

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

URI: "http://example.com"
Title: "Longest day of year"
Text: "The summer solstice is on Friday."
```

You can also embed SQL where clauses in queries:

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

which produces this output:

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

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

## Database Wrap Up

You learned the basics for accessing relational databases. When I am designing new systems for processing data I like to think of my Common Lisp code as being purely functional: my Lisp functions accept arguments that they do not modify and return results. I like to avoid side effects, that is changing global state. When I do have to handle mutable state (or data) I prefer storing mutable state in an external database. I use this same approach when I use the Haskell functional programming language.

# Using MongoDB, Solr NoSQL Data Stores

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

The MongoDB example code is in the file **src/loving_snippets/mongo_news.lisp**. The Solr example code is in the subdirectories **src/solr_examples**.

**Note for the fifth edition**: The Common Lisp cl-mongo library is now unsupported for versions of MongoDB later than 2.6 (released in 2016). You can install an old version of MongoDB for macOS[37] or for Linux[38]. I have left the MongoDB examples in this section but I can't recommend that you use cl-mongo and MongoDB for any serious applications.

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**vailablity 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.

## 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

[37] https://www.mongodb.org/dl/osx
[38] https://www.mongodb.org/dl/linux

(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 usually performing 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 it 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[39] 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[40].

At this time there is no official Common Lisp support for accessing MongoDB but there is a useful project by Alfons Haffmans' cl-mongo[41] 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:

[39] http://docs.mongodb.org/manual/
[40] https://www.mongohq.com/
[41] https://github.com/fons/cl-mongo

```
(ql:quickload "cl-mongo")

(cl-mongo:db.use "news")

(defun add-article (uri title text)
  (let ((doc (cl-mongo:make-document)))
        (cl-mongo:add-element "uri" uri doc)
        (cl-mongo:add-element "title" title doc)
        (cl-mongo:add-element "text" text doc)
        (cl-mongo:db.insert "article" doc)))

;; add a test document:
(add-article "http://test.com" "article title 1" "article text 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 fetchng 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:

```
(defun print-articles ()
  (cl-mongo:pp (cl-mongo:iter (cl-mongo:db.find "article" :all))))

;; for each document, use the cl-mongo:get-element on
;; each element we want to save:
(defun article-results->lisp-data (mdata)
  (let ((ret '()))
    ;;(print (list "size of result=" (length mdata)))
    (dolist (a mdata)
      ;;(print a)
      (push
        (list
         (cl-mongo:get-element "uri" a)
         (cl-mongo:get-element "title" a)
         (cl-mongo:get-element "text" a))
        ret)))
    ret))

(defun get-articles ()
  (article-results->lisp-data
    (cadr (cl-mongo:db.find "article" :all))))
```

Output for these two functions looks like:

```
* (print-articles)

{
  "_id" -> objectid(99778A792EBB4F76B82F75C6)
  "uri"  ->  http://test.com/3
  "title"  ->  article title 3
  "text"  ->  article text 3
}

{
  "_id" -> objectid(D47DEF3CFDB44DEA92FD9E56)
  "uri"  ->  http://test.com/2
  "title"  ->  article title 2
  "text"  ->  article text 2
}

* (get-articles)

(("http://test.com/2" "article title 2" "article text 2")
 ("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:

```
;; find documents where substring 'str' is in the title:
(defun search-articles-title (str)
  (article-results->lisp-data
    (cadr
      (cl-mongo:iter
       (cl-mongo:db.find
         "article"
         (cl-mongo:kv
            "title"     // TITLE ATTRIBUTE
            (cl-mongo:kv "$regex" str)) :limit 10)))))

;; find documents where substring 'str' is in the text element:
(defun search-articles-text (str)
  (article-results->lisp-data
    (cadr
      (cl-mongo:db.find
```

```
        "article"
        (cl-mongo:kv
           "text"        // TEXT ATTRIBUTE
           (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**:

```
* (SEARCH-ARTICLES-TEXT "text")

(("http://test.com/2" "article title 2" "article text 2")
 ("http://test.com/3" "article title 3" "article text 3"))
* (SEARCH-ARTICLES-TEXT "3")

(("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[42], and easy to use client libraries like **clouchdb** 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:

```
->  src git:(master) mongo
MongoDB shell version: 2.4.5
connecting to: test
> show dbs
kbsportal        0.03125GB
knowledgespace        0.03125GB
local        (empty)
mark_twitter        0.0625GB
myfocus        0.03125GB
news        0.03125GB
nyt        0.125GB
twitter        0.125GB
> use news
switched to db news
> show collections
article
system.indexes
```

[42] http://docs.mongodb.org/manual/mongo/

```
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[43] 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/[44].

# 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 to count them. The Apache Solr Project[45] 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 embed 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.

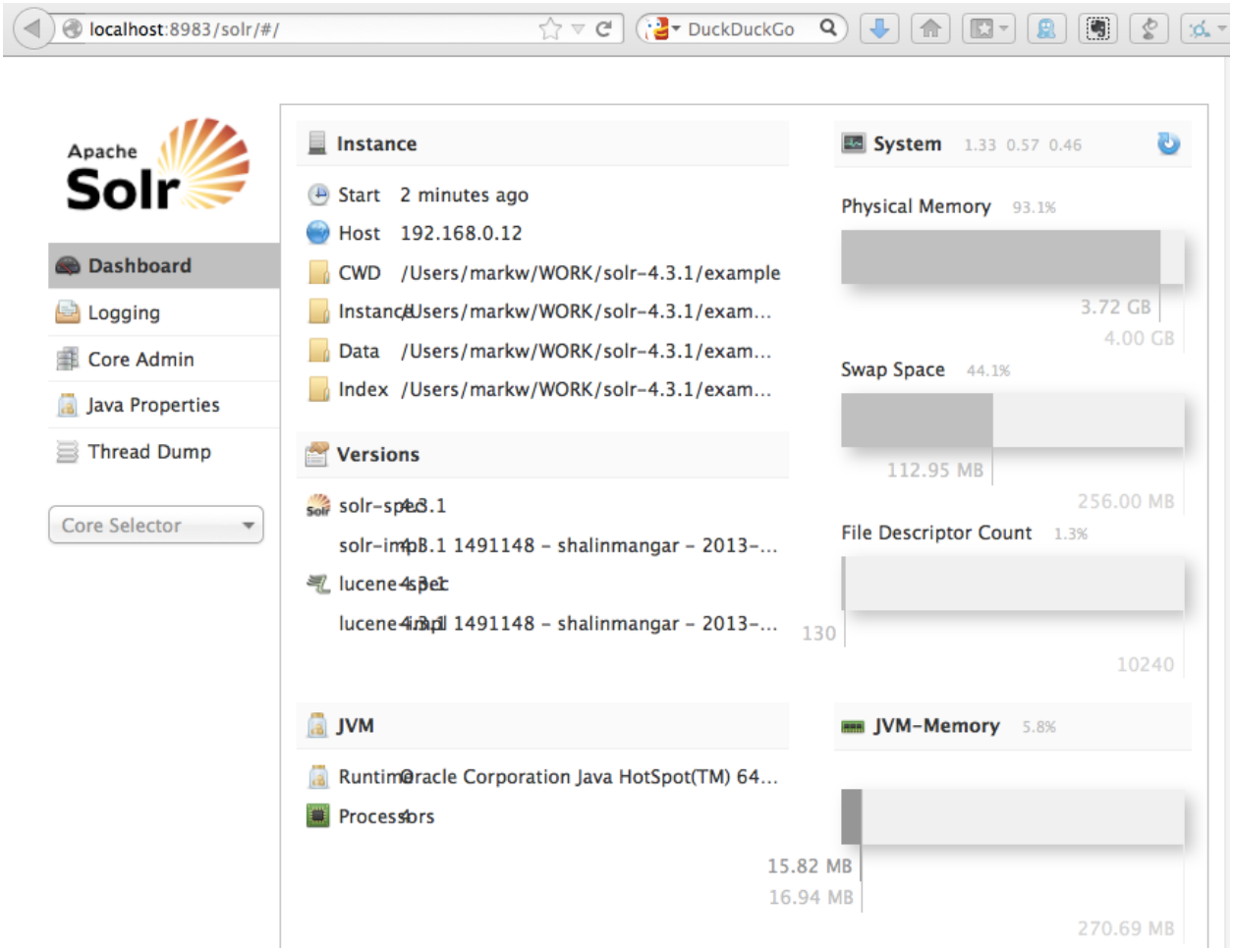
## Installing Solr

Download a binary Solr distribution[46] and un-tar or un-zip this Solr distribution, cd to the distribution directory, then cd to the example directory and run:

[43] http://docs.mongodb.org/manual/mongo/
[44] http://localhost:28017/
[45] https://lucene.apache.org/solr/
[46] https://lucene.apache.org/solr/downloads.html

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

You can access the Solr Admin Web App at http://localhost:8983/solr/#/[47]. 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:

[47] http://localhost:8983/solr/#/

```
 ~/> cd ~/solr/example/exampledocs
 ~/solr/example/exampledocs> java -jar post.jar *.xml
SimplePostTool version 1.5
Posting files to base url http://localhost:8983/solr/update
        using content-type application/xml..
POSTing file gb18030-example.xml
POSTing file hd.xml
POSTing file ipod_other.xml
POSTing file ipod_video.xml
POSTing file manufacturers.xml
POSTing file mem.xml
POSTing file money.xml
POSTing file monitor.xml
POSTing file monitor2.xml
POSTing file mp500.xml
POSTing file sd500.xml
POSTing file solr.xml
POSTing file utf8-example.xml
POSTing file vidcard.xml
14 files indexed.
COMMITting Solr index changes
           to http://localhost:8983/solr/update..
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 that 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.

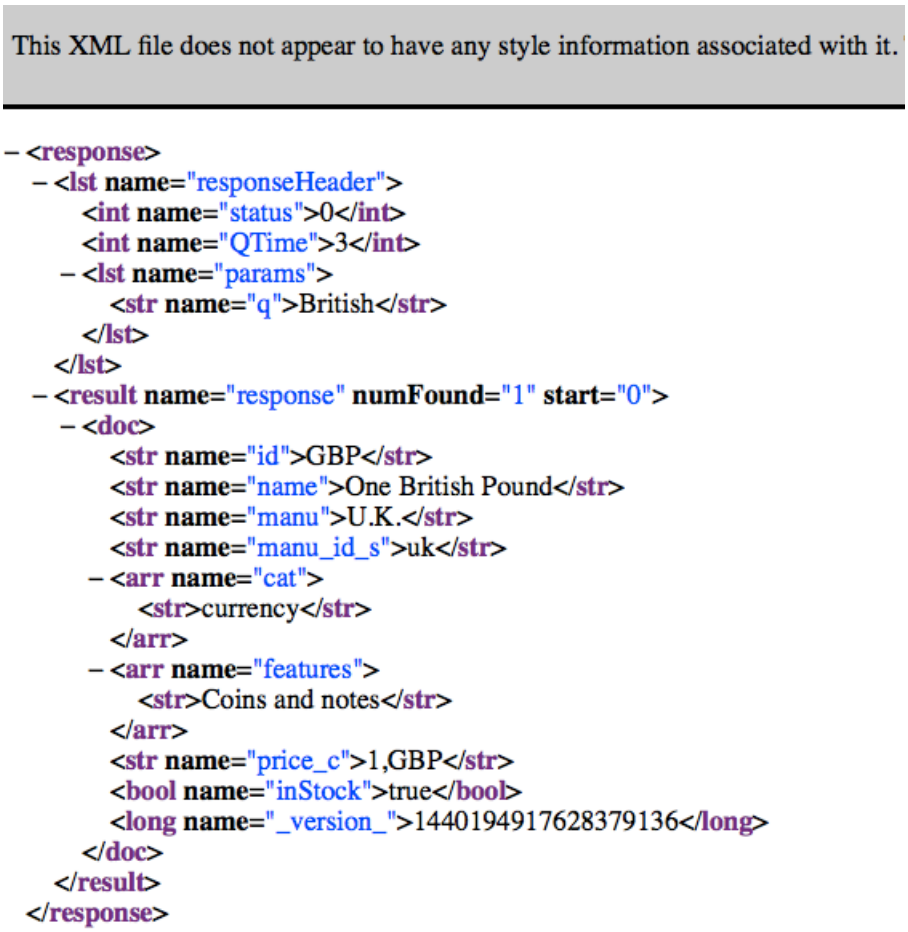
## Solr's REST Interface

The Solr REST Interface Documentation[48] 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[49]. This is a REST request URL and you can use utilities like **curl** or **wget** to fetch the XML data. I fetched

[48] https://wiki.apache.org/solr/SolJSON
[49] http://localhost:8983/solr/select?q=British

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):




**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:

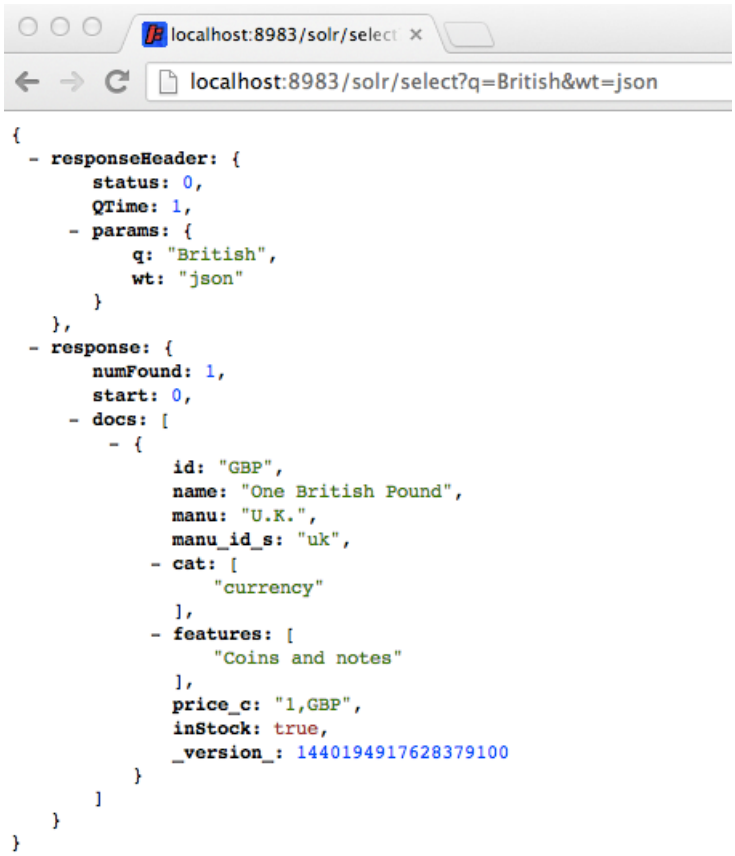
```
<doc>
  <field name="id">GBP</field>
  <field name="name">One British Pound</field>
  <field name="manu">U.K.</field>
  <field name="manu_id_s">uk</field>
  <field name="cat">currency</field>
  <field name="features">Coins and notes</field>
  <field name="price_c">1,GBP</field>
  <field name="inStock">true</field>
</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. If for example we were indexing news stories, then example input data might look like:

```
<doc>
  <field name="id">new_story_0001</field>
  <field name="title">Fishing Season Opens</field>
  <field name="text">Fishing season opens on Friday in Oak Creek.</field>
</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):




**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 sawearlier 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:

```
(ql:quickload :drakma)
(ql:quickload :cl-json)

(defun do-search (&rest terms)
  (let ((query-string (format nil "~{~A~^+AND+~}" terms)))
   (cl-json:decode-json-from-string
     (drakma:http-request
       (concatenate
        'string
        "http://localhost:8983/solr/select?q="
        query-string
        "&wt=json")))))
```

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

```
* (do-search "British" "one")

((:RESPONSE-HEADER (:STATUS . 0) (:*Q-TIME . 1)
  (:PARAMS (:Q . "British+AND+one") (:WT . "json")))
 (:RESPONSE (:NUM-FOUND . 6) (: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") (:IN-STOCK . T)
    (:--VERSION-- . 1440194917628379136))
   ((:ID . "USD") (:NAME . "One Dollar")
    (:MANU . "Bank of America")
    (:MANU--ID--S . "boa") (:CAT "currency")
    (:FEATURES "Coins and notes")
    (:PRICE--C . "1,USD") (:IN-STOCK . T)
    (:--VERSION-- . 1440194917624184832))
   ((:ID . "EUR") (:NAME . "One Euro")
    (:MANU . "European Union")
    (:MANU--ID--S . "eu") (:CAT "currency")
    (:FEATURES "Coins and notes")
    (:PRICE--C . "1,EUR") (:IN-STOCK . T)
    (:--VERSION-- . 1440194917626281984))
   ((:ID . "NOK") (:NAME . "One Krone")
    (:MANU . "Bank of Norway")
    (:MANU--ID--S . "nor") (:CAT "currency")
    (:FEATURES "Coins and notes")
    (:PRICE--C . "1,NOK") (:IN-STOCK . T)
    (:--VERSION-- . 1440194917631524864))
   ((:ID . "0579B002")
    (:NAME . "Canon PIXMA MP500 All-In-One Photo Printer")
    (:MANU . "Canon Inc.")
    (:MANU--ID--S . "canon")
    (:CAT "electronics" "multifunction printer"
     "printer" "scanner" "copier")
    (:FEATURES "Multifunction ink-jet color photo printer"
     "Flatbed scanner, optical scan resolution of 1,200 x 2,400 dpi"
     "2.5\" color LCD preview screen" "Duplex Copying"
     "Printing speed up to 29ppm black, 19ppm color" "Hi-Speed USB"
     "memory card: CompactFlash, Micro Drive, SmartMedia,
      Memory Stick, Memory Stick Pro, SD Card, and MultiMediaCard")
    (:WEIGHT . 352.0) (:PRICE . 179.99)
    (:PRICE--C . "179.99,USD")
```

```
    (:POPULARITY . 6) (:IN-STOCK . T)
    (:STORE . "45.19214,-93.89941")
    (:--VERSION-- . 1440194917651447808))
   ((:ID . "SOLR1000")
    (:NAME . "Solr, the Enterprise Search Server")
    (:MANU . "Apache Software Foundation")
    (:CAT "software" "search")
    (:FEATURES "Advanced Full-Text Search Capabilities using Lucene"
     "Optimized for High Volume Web Traffic"
     "Standards Based Open Interfaces - XML and HTTP"
     "Comprehensive HTML Administration Interfaces"
     "Scalability - Efficient Replication to other Solr Search Servers"
     "Flexible and Adaptable with XML configuration and Schema"
     "Good unicode support: hÃ©llo (hello with an accent over the e)")
    (:PRICE . 0.0) (:PRICE--C . "0,USD") (:POPULARITY . 10) (:IN-STOCK . T)
    (:INCUBATIONDATE--DT . "2006-01-17T00:00:00Z")
    (:--VERSION-- . 1440194917671370752)))))
```

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

```
* (cdr (cadddr (cadr (do-search "British" "one"))))

(((:ID . "GBP") (:NAME . "One British Pound") (:MANU . "U.K.")
  (:MANU--ID--S . "uk") (:CAT "currency") (:FEATURES "Coins and notes")
  (:PRICE--C . "1,GBP") (:IN-STOCK . T)
  (:--VERSION-- . 1440194917628379136))
 ((:ID . "USD") (:NAME . "One Dollar") (:MANU . "Bank of America")
  (:MANU--ID--S . "boa") (:CAT "currency") (:FEATURES "Coins and notes")
  (:PRICE--C . "1,USD") (:IN-STOCK . T)
  (:--VERSION-- . 1440194917624184832))
 ((:ID . "EUR") (:NAME . "One Euro") (:MANU . "European Union")
  (:MANU--ID--S . "eu") (:CAT "currency") (:FEATURES "Coins and notes")
  (:PRICE--C . "1,EUR") (:IN-STOCK . T)
  (:--VERSION-- . 1440194917626281984))
 ((:ID . "NOK") (:NAME . "One Krone") (:MANU . "Bank of Norway")
  (:MANU--ID--S . "nor") (:CAT "currency")
  (:FEATURES "Coins and notes")
  (:PRICE--C . "1,NOK") (:IN-STOCK . T)
  (:--VERSION-- . 1440194917631524864))
 ((:ID . "0579B002")
  (:NAME . "Canon PIXMA MP500 All-In-One Photo Printer")
  (:MANU . "Canon Inc.") (:MANU--ID--S . "canon")
```

```
  (:CAT "electronics" "multifunction printer" "printer"
   "scanner" "copier")
  (:FEATURES "Multifunction ink-jet color photo printer"
   "Flatbed scanner, optical scan resolution of 1,200 x 2,400 dpi"
   "2.5\" color LCD preview screen" "Duplex Copying"
   "Printing speed up to 29ppm black, 19ppm color" "Hi-Speed USB"
   "memory card: CompactFlash, Micro Drive, SmartMedia, Memory Stick,
    Memory Stick Pro, SD Card, and MultiMediaCard")
  (:WEIGHT . 352.0) (:PRICE . 179.99) (:PRICE--C . "179.99,USD")
  (:POPULARITY . 6) (:IN-STOCK . T) (:STORE . "45.19214,-93.89941")
  (:--VERSION-- . 1440194917651447808))
 ((:ID . "SOLR1000") (:NAME . "Solr, the Enterprise Search Server")
  (:MANU . "Apache Software Foundation") (:CAT "software" "search")
  (:FEATURES "Advanced Full-Text Search Capabilities using Lucene"
   "Optimized for High Volume Web Traffic"
   "Standards Based Open Interfaces - XML and HTTP"
   "Comprehensive HTML Administration Interfaces"
   "Scalability - Efficient Replication to other Solr Search Servers"
   "Flexible and Adaptable with XML configuration and Schema"
   "Good unicode support: hÃ©llo (hello with an accent over the e)")
  (:PRICE . 0.0) (:PRICE--C . "0,USD") (:POPULARITY . 10) (:IN-STOCK . T)
  (:INCUBATIONDATE--DT . "2006-01-17T00:00:00Z")
  (:--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:

```
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:

```
<add>
    <doc>
        <field name="id">123456</field>
        <field name="title">Fishing Season</field>
    </doc>
</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:

```
(do-add '(("id" . "12345")
          ("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:

```
(defun commit-adds ()
  (drakma:http-request
   "http://localhost:8983/solr/update"
   :method :post
   :content-type "application/xml"
   :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:

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

200
((:CONTENT-TYPE . "application/xml; charset=UTF-8")
 (:CONNECTION . "close"))
#<PURI:URI http://localhost:8983/solr/update>
#<FLEXI-STREAMS:FLEXI-IO-STREAM {1009193133}>
T
"OK"
* (commit-adds)

200
((:CONTENT-TYPE . "application/xml; charset=UTF-8")
```

```
 (:CONNECTION . "close"))
#<PURI:URI http://localhost:8983/solr/update>
#<FLEXI-STREAMS:FLEXI-IO-STREAM {10031F20B3}>
T
"OK"
* (do-search "fishing")

((:RESPONSE-HEADER (:STATUS . 0) (:*Q-TIME . 2)
  (:PARAMS (:Q . "fishing") (:WT . "json")))
 (:RESPONSE (:NUM-FOUND . 1) (:START . 0)
  (:DOCS
   ((:ID . "12345\"") (:TITLE "Fishing Season\"")
    (:--VERSION-- . 1440293991717273600)))))
*
```

## 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. 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.

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 (POS tagging) 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.)

We will use a library that I wrote that performs POS tagging, categorization (classification), summarization, and detects proper names.

My example code for this chapter is contained in separate Quicklisp projects located in the subdirectories:

- **src/fasttag**: performs part of speech tagging and tokenizes text
- **src/categorize_summarize**: performs categorization (e.g., detects the topic of text is news, politics, economy, etc.) and text summarization
- **src/kbnlp**: the top level APIs for my pure Common Lisp natural language processing (NLP) code. In later chapters we will take a different approach by using Python deep learning models for NLP that we call as a web service. I use both approaches in my own work.

I worked on this Lisp code, and also similar code in Java, from about 2001 to 2011, and again in 2019 for my application for generating knowledge graph data automatically (this is an example in a later chapter). I am going to begin 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 for categorization (word counts) 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/linguistic_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/kbnlp** directory:

```
%  loving-common-lisp git:(master) > cd src/kbnlp
%  src/kbnlp git:(master) > sbcl
* (ql:quickload "kbnlp")

"Startng to load data...."
"....done loading data."
*
```

This also loads the projects in **src/fasttag** and **src/categorize_summarize**.

Unfortunately, it takes about a minute using SBCL 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.

Note: I often use Clozure-CL (CCL) instead of SBCL for developing my NLP libraries because CCL loads my data files much faster than SBCL.

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
```

```
*X*

* *X*

#S(TEXT
   :URL ""
   :TITLE ""
   :SUMMARY "<no summary>"
   :CATEGORY-TAGS (("news_politics.txt" 0.01648)
                   ("news_economy.txt" 0.01601))
   :KEY-WORDS NIL
   :KEY-PHRASES NIL
   :HUMAN-NAMES ("President Bob Smith")
   :PLACE-NAMES NIL
   :TEXT #("President" "Bob" "Smith" "talked" "to" "Congress" "about" "the"
           "economy" "and" "taxes")
   :TAGS #("NNP" "NNP" "NNP" "VBD" "TO" "NNP" "IN" "DT" "NN" "CC" "NNS")
   :STEMS #("presid" "bob" "smith" "talk" "to" "congress" "about" "the"
            "economi" "and" "tax"))
*
```

At the end of the file **src/knowledgebooks_nlp.lisp** 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:

```
(:SUMMARY
  "Often those amendments are an effort to change government policy
   by adding or subtracting money for carrying it out. The initial
   surge in foreclosures in 2007 and 2008 was tied to subprime
   mortgages issued during the housing boom to people with shaky
   credit. 2 trillion in annual appropriations bills for funding
   most government programs – usually low profile legislation that
   typically dominates the work of the House in June and July.
   Bill Clinton said that banking in Europe is a good business.
   These days homeowners who got fixed rate prime mortgages because
   they had good credit cannot make their payments because they are
   out of work. The question is whether or not the US dollar remains
   the world s reserve currency if not the US economy will face
   a depression."
:CATEGORY-TAGS (("news_politics.txt" 0.38268)
                ("news_economy.txt" 0.31182)
                ("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:

```
(defstruct text
  url
  title
  summary
  category-tags
  key-words
  key-phrases
  human-names
  place-names
  text
  tags
  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.

If you just need part of speech tagging (and not summarization, categorization, and top level APIs used in the last section) you can load:

```
(ql:quickload "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:

```
:TEXT #("President" "Bob" "Smith" "talked" "to" "Congress" "about" "the"
        "economy" "and" "taxes")
: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 Conjuncn | 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 | ( |
| VBN | 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 tagged as common nouns, but end in "ing" so they should be a gerand (verb form):

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

You can find the lexicon data in the file **src/linguistic_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:

```
(defvar lex-hash (make-hash-table :test #'equal :size 110000))
(setf (gethash "shakeup" lex-hash) (list "NN"))
(setf (gethash "Laurance" lex-hash) (list "NNP"))
(setf (gethash "expressing" lex-hash) (list "VBG"))
(setf (gethash "citybred" lex-hash) (list "JJ"))
(setf (gethash "negative" lex-hash) (list "JJ" "NN"))
(setf (gethash "investors" lex-hash) (list "NNS" "NNPS"))
(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:

If you only need categorization and not the other libraries developed in this chapter, you can just load this library and run the example in the comment at the bottom of the file **categorize_summarize.lisp**:

({lang="lisp",linenos=off} (ql:quickload "categorize_summarize") (defvar x "President Bill Clinton <<2 pages text no shown>> ") (defvar words1 (myutils:words-from-string x)) (print words1) (setq cats1 (categorize_summarize:categorize words1)) (print cats1) (defvar sum1 (categorize_summarize:summarize words1 cats1)) (print sum1)

Let's look at the implementation, starting with creating hash tables for storing word count data for each category or topic:

```
;;;  Starting topic: news_economy.txt

(setf *h* (make-hash-table :test #'equal :size 1000))

  (setf (gethash "news" *h*) 3915)
  (setf (gethash "debt" *h*) 3826)
  (setf (gethash "money" *h*) 1809)
  (setf (gethash "work" *h*) 1779)
  (setf (gethash "business" *h*) 1631)
  (setf (gethash "tax" *h*) 1572)
  (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:

```
(defun get-word-list-category (words)
  (let ((x nil)
        (ss nil)
        (cat-hash nil)
        (word nil)
        (len (length words))
        (num-categories (length categoryHashtables))
        (category-score-accumulation-array
          (make-array num-categories :initial-element 0)))

    (defun list-sort (list-to-sort)
      ;;(pprint list-to-sort)
      (sort list-to-sort
            #'(lambda (list-element-1 list-element-2)
                (> (cadr list-element-1) (cadr list-element-2)))))

    (do ((k 0 (+ k 1)))
        ((equal k len))
      (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 i))))))
28      (setf ss '())
29      (do ((i 0 (+ i 1)))
30          ((equal i num-categories))
31        (if (> (aref category-score-accumulation-array i) 0.01)
32            (setf
33              ss
34              (cons
35                (list
36                  (nth i categoryNames)
37                  (round (* (aref category-score-accumulation-array i) 10)))
38                ss))))
39      (setf ss (list-sort ss))
40      (let ((cutoff (/ (cadar ss) 2))
41            (results-array '()))
42        (dolist (hit ss)
43          (if (> (cadr hit) cutoff)
44              (setf results-array (cons hit results-array))))
45        (reverse results-array))))
```

On thing to notice in this listing is lines 11 through 15 where I define a nested function **list-sort** that takes a list of sub-lists 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.

## Detecting People's Names and Place Names

The code for detecting people and place names is in the top level API code in the package defined in **src/kbnlp**. This package is loaded using:

```
(ql:quickload "kbnlp")
(kbnlp:make-text-object "President Bill Clinton ran for president of the USA")
```

The functions that support identifying people's names and place names in text are in the Common Lisp package **kb nlp**::

- 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 a 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 **defstruct text** object as input and modifies the **defstruct 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) :test #'equal))
9           (place-list
10            (remove-duplicates
11              (build-list-find-name-helper words place-indices) :test #'equal)))
12     (let ((ret '()))
13       (dolist (x name-list)
14         (if (search " " x)
15             (setq ret (cons x ret))))
16       (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

The code for summarizing text is located in the directory **src/categorize_summarize** and can be loaded using:

({lang="lisp",linenos=off} (ql:quickload "categorize_summarize")

The code for summarization depends on the categorization code we saw earlier.

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. 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.

```
(defun summarize (txt-obj)
  (let* ((words (text-text txt-obj))
         (num-words (length words))
         (cats (text-category-tags txt-obj))
         (sentence-count 0)
         best-sentences sentence (score 0))
    ;; loop over sentences:
    (dotimes (i num-words)
      (let ((word (svref words i)))
        (dolist (cat cats)
          (let* ((hash (gethash (car cat) categoryToHash))
                 (value (gethash word hash)))
            (if value
                (setq score (+ score (* 0.01 value (cadr cat))))))))
        (push word sentence)
        (if (or (equal word ".") (equal word "!") (equal word ";"))
            (let ()
              (setq sentence (reverse sentence))
              (setq score (/ score (1+ (length sentence))))
              (setq sentence-count (1+ sentence-count))
              (format t "~%~A : ~A~%" sentence score)
              ;; process this sentence:
              (if (and
                   (> score 0.4)
                   (> (length sentence) 4)
                   (< (length sentence) 30))
                  (progn
                    (setq sentence
                          (reduce
                           #'(lambda (x y) (concatenate 'string x " " y))
                           (coerce sentence 'list)))
                    (push (list sentence score) best-sentences)))
              (setf sentence nil score 0)))))
    (setf
     best-sentences
     (sort
      best-sentences
      #'(lambda (x y) (> (cadr x) (cadr y)))))
    (if best-sentences
        (replace-all
         (reduce #'(lambda (x y) (concatenate 'string x " " y))
                 (mapcar #'(lambda (x) (car x)) best-sentences))
         " ." ".")
```

```
44          "<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/kbnlp/linguistic_data/names/names.companies** and **src/kbnlp/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[50] 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[51] 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.[52] 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.

To load and run an example, try:

```
(ql:quickload "dbpedia")
(dbpedia:dbpedia-lookup "berlin")
```

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[53] using the DBPedia Lookup Service in the next code listing that shows the contents of the example file **dbpedia-lookup.lisp** in the directory **src/dbpedia**:

[50] http://markwatson.com/#books/
[51] https://en.wikipedia.org/wiki/Wikipedia:Database_download
[52] http://download.wikimedia.org/enwiki/latest/enwiki-latest-pages-articles.xml.bz2
[53] http://dbpedia.org/

```
(ql:quickload :drakma)
(ql:quickload :babel)
(ql:quickload :s-xml)

;; utility from http://cl-cookbook.sourceforge.net/strings.html#manip:
(defun replace-all (string part replacement &key (test #'char=))
  "Returns a new string in which all the occurrences of the part
is replaced with replacement."
  (with-output-to-string (out)
    (loop with part-length = (length part)
       for old-pos = 0 then (+ pos part-length)
       for pos = (search part string
                         :start2 old-pos
                         :test test)
       do (write-string string out
                        :start old-pos
                        :end (or pos (length string)))
       when pos do (write-string replacement out)
       while pos)))

(defstruct dbpedia-data uri label description)

(defun dbpedia-lookup (search-string)
  (let* ((s-str (replace-all search-string " " "+"))
         (s-uri
          (concatenate
           'string
           "http://lookup.dbpedia.org/api/search.asmx/KeywordSearch?QueryString="
           s-str))
         (response-body nil)
         (response-status nil)
         (response-headers nil)
         (xml nil)
         ret)
    (multiple-value-setq
        (response-body response-status response-headers)
      (drakma:http-request
       s-uri
       :method :get
       :accept "application/xml"))
    ;; (print (list "raw response body as XML:" response-body))
    ;;(print (list ("status:" response-status "headers:" response-headers)))
    (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:dbpedia-lookup** (only some of the returned data is shown):

```
1  * (ql:quickload "dbpedia")
2  * (dbpedia: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[54] 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/dbpedia-lookup.lisp** (covered in the last section) and **src/solr_examples/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[55] and I recommend that as a starting point.

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.

[54] http://dbpedia.org
[55] http://articulate-lisp.com/examples/trotter.html

# Using Apache Nutch

Apache Nutch[56], 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_examples/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[57] 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** a good start is OpenSearch services that return JSON data and OpenSearch Community JSON formats web page[58] 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.

[56] https://nutch.apache.org/
[57] http://www.opensearch.org/Home
[58] http://www.opensearch.org/Community/JSON_Formats

## 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.

# Using The CL Machine-Learning Library

The CL Machine-Learning (CLML) library was originally developed by MSI (NTT DATA Mathematical Systems Inc. in Japan) and is supported by many developers. You should visit the CLML web page[59] for project documentation and follow the installation directions and read about the project before using the examples in this chapter. However if you just want to quickly try the following CLML examples then you can install CLML using Quicklisp:

```
mkdir -p ~/quicklisp/local-projects
cd ~/quicklisp/local-projects
git clone https://github.com/mmaul/clml.git
sbcl --dynamic-space-size 2560
> (ql:quickload :clml :verbose t)
```

The installation will take a while to run but after installation using the libraries via quickload is fast. You can now run the example Quicklisp project **src/clml_examples**:

```
$ sbcl --dynamic-space-size 2560
* (ql:quickload "clmltest")
* (clmltest:clml-tests-example)
```

Please be patient the first time you run this because the first time you load the example project, the one time installation of CLML will take a while to run but after installation then the example project loads quickly. CLML installation involves downloading and installing BLAS, LAPACK, and other libraries.

Other resources for CLML are the tutorials[60] and contributed extensions[61] that include support for plotting (using several libraries) and for fetching data sets.

Although CLML is fairly portable we will be using SBCL and we need to increase the heap space when starting SBCL when we want to use the CLML library:

```
sbcl --dynamic-space-size 5000
```

[59] https://github.com/mmaul/clml
[60] https://github.com/mmaul/clml.tutorials
[61] https://github.com/mmaul/clml.extras

You can refer to the documentation at https://github.com/mmaul/clml[62]. This documentation lists the packages with some information for each package but realistically I keep the source code for CLML in an editor or IDE and read source code while writing code that uses CLML. I will show you with short examples how to use the KNN (K nearest neighbors) and SVM (support vector machines) APIs. We will not cover other useful CLML APIs like time series processing, Naive Bayes, PCA (principle component analysis) and general matrix and tensor operations.

Even though the learning curve is a bit steep, CLML provides a lot of functionality for machine learning, dealing with time series data, and general matrix and tensor operations.

# Using the CLML Data Loading and Access APIs

The CLML project uses several data sets and since the few that we will use are small files, they are included in the book's repository in directory *machine_learning_data* under the *src* directory. The first few lines of *labeled_cancer_training_data.csv* are:

```
Cl.thickness,Cell.size,Cell.shape,Marg.adhesion,Epith.c.size,Bare.nuclei,Bl.cromatin\
,Normal.nucleoli,Mitoses,Class
5,4,4,5,7,10,3,2,1,benign
6,8,8,1,3,4,3,7,1,benign
8,10,10,8,7,10,9,7,1,malignant
2,1,2,1,2,1,3,1,1,benign
```

The first line in the CSV data files specifies names for each attribute with the name of the last column being "Class" which here takes on values *benign* or *malignant*. Later, the goal will be to create models that are constructed from training data and then make predictions of the "Class" of new input data. We will look at how to build and use machine learning models later but here we concentrate on reading and using input data.

The example file *clml_data_apis.lisp* shows how to open a file and loop over the values for each row:

```
1  ;; note; run SBCL using: sbcl --dynamic-space-size 2560
2
3  (ql:quickload '(:clml
4                  :clml.hjs)) ; read data sets
5
6  (defpackage #:clml-data-test
7    (:use #:cl #:clml.hjs.read-data))
8
9  (in-package #:clml-data-test)
```

[62] https://github.com/mmaul/clml

```
10
11 (defun read-data ()
12   (let ((train1
13          (clml.hjs.read-data:read-data-from-file
14           "./machine_learning_data/labeled_cancer_training_data.csv"
15           :type :csv
16           :csv-type-spec (append
17                           (make-list 9 :initial-element 'double-float)
18                           '(symbol)))))
19     (loop-over-and-print-data train1)))
20
21 (defun loop-over-and-print-data (clml-data-set)
22   (print "Loop over and print a CLML data set:")
23   (let ((testdata (clml.hjs.read-data:dataset-points clml-data-set)))
24     (loop for td across testdata
25        do
26          (print td))))
27
28 (read-data)
```

The function **read-data** defined in lines 11-19 uses the utility function **clml.hjs.read-data:read-data-from-file** to read a CSV (comma separated value) spreadsheet file from disk. The CSV file is expected to contain 10 columns (set in lines 17-18) with the first nine columns containing floating point values and the last column text data.

The function **loop-over-and-print-data** defined in lines 21-26 reads the CLML data set object, looping over each data sample (i.e., each row in the original spreadsheet file) and printing it.

Here is some output from loading this file:

```
1  $ sbcl --dynamic-space-size 2560
2  This is SBCL 1.3.16, an implementation of ANSI Common Lisp.
3  More information about SBCL is available at <http://www.sbcl.org/>.
4
5  SBCL is free software, provided as is, with absolutely no warranty.
6  It is mostly in the public domain; some portions are provided under
7  BSD-style licenses.  See the CREDITS and COPYING files in the
8  distribution for more information.
9  * (load "clml_data_apis.lisp")
10
11 "Loop over and print a CLML data set:"
12 #(5.0d0 4.0d0 4.0d0 5.0d0 7.0d0 10.0d0 3.0d0 2.0d0 1.0d0 |benign|)
13 #(6.0d0 8.0d0 8.0d0 1.0d0 3.0d0 4.0d0 3.0d0 7.0d0 1.0d0 |benign|)
```

```
#(8.0d0 10.0d0 10.0d0 8.0d0 7.0d0 10.0d0 9.0d0 7.0d0 1.0d0 |malignant|)
#(2.0d0 1.0d0 2.0d0 1.0d0 2.0d0 1.0d0 3.0d0 1.0d0 1.0d0 |benign|)
```

In the next section we will use the same cancer data training file, and another test data in the same format to cluster this cancer data into similar sets, one set for non-malignant and one for malignant samples.

## K-Means Clustering of Cancer Data Set

We will now read the same University of Wisconsin cancer data set and cluster the input samples (one sample per row of the spreadsheet file) into similar classes. We will find after training a model that the data is separated into two clusters, representing non-malignant and malignant samples.

The function **cancer-data-cluster-example-read-data** defined in lines 33-47 is very similar to the function **read-data** in the last section except here we read in two data files: one for training and one for testing.

The function **cluster-using-k-nn** defined in lines 13-30 uses the training and test data objects to first train a model and then to test it with test data that was previously used for training. Notice how we call this function in line 47: the first two arguments are the two data set objects, the third is the string "Class" that is the label for the 10th column of the original spreadsheet CSV files, and the last argument is the type of distance measurement used to compare two data samples (i.e., comparing any two rows of the training CSV data file).

```
;; note; run SBCL using: sbcl --dynamic-space-size 2560

(ql:quickload '(:clml
                :clml.hjs ; utilities
                :clml.clustering))

(defpackage #:clml-knn-cluster-example1
  (:use #:cl #:clml.hjs.read-data))

(in-package #:clml-knn-cluster-example1)

;; folowing is derived from test code in CLML:
(defun cluster-using-k-nn (test train objective-param-name  manhattan)
  (let (original-data-column-length)
    (setq original-data-column-length
          (length (aref (clml.hjs.read-data:dataset-points train) 0)))
    (let* ((k 5)
           (k-nn-estimator
            (clml.nearest-search.k-nn:k-nn-analyze train
```

```
              k
              objective-param-name :all
              :distance manhattan :normalize t)))
      (loop for data across
          (dataset-points
            (clml.nearest-search.k-nn:k-nn-estimate k-nn-estimator test))
        if (equal (aref data 0) (aref data original-data-column-length))
        do
          (format t "Correct: ~a~%" data)
        else do
          (format t "Wrong:   ~a~%" data)))))

;; folowing is derived from test code in CLML:
(defun cancer-data-cluster-example-read-data ()
  (let ((train1
         (clml.hjs.read-data:read-data-from-file
          "./machine_learning_data/labeled_cancer_training_data.csv"
          :type :csv
          :csv-type-spec (append (make-list 9 :initial-element 'double-float)
                                 '(symbol))))
        (test1
         (clml.hjs.read-data:read-data-from-file
          "./machine_learning_data/labeled_cancer_test_data.csv"
          :type :csv
          :csv-type-spec (append (make-list 9 :initial-element 'double-float)
                                 '(symbol)))))
    ;;(print test1)
    (print (cluster-using-k-nn test1 train1 "Class" :double-manhattan))))

(cancer-data-cluster-example-read-data)
```

The following listing shows the output from running the last code example:

```
Number of self-misjudgement : 13
Correct: #(benign 5.0d0 1.0d0 1.0d0 1.0d0 2.0d0 1.0d0 3.0d0 1.0d0 1.0d0 benign)
Correct: #(benign 3.0d0 1.0d0 1.0d0 1.0d0 2.0d0 2.0d0 3.0d0 1.0d0 1.0d0 benign)
Correct: #(benign 4.0d0 1.0d0 1.0d0 3.0d0 2.0d0 1.0d0 3.0d0 1.0d0 1.0d0 benign)
Correct: #(benign 1.0d0 1.0d0 1.0d0 1.0d0 2.0d0 10.0d0 3.0d0 1.0d0 1.0d0 benign)
Correct: #(benign 2.0d0 1.0d0 1.0d0 1.0d0 2.0d0 1.0d0 1.0d0 1.0d0 5.0d0 benign)
Correct: #(benign 1.0d0 1.0d0 1.0d0 1.0d0 1.0d0 1.0d0 3.0d0 1.0d0 1.0d0 benign)
Wrong:   #(benign 5.0d0 3.0d0 3.0d0 3.0d0 2.0d0 3.0d0 4.0d0 4.0d0 1.0d0
           malignant)
Correct: #(malignant 8.0d0 7.0d0 5.0d0 10.0d0 7.0d0 9.0d0 5.0d0 5.0d0 4.0d0
```

```
11            malignant)
12  Correct: #(benign 4.0d0 1.0d0 1.0d0 1.0d0 2.0d0 1.0d0 2.0d0 1.0d0 1.0d0 benign)
13  Correct: #(malignant 10.0d0 7.0d0 7.0d0 6.0d0 4.0d0 10.0d0 4.0d0 1.0d0 2.0d0
14            malignant)
15   ...
```

## SVM Classification of Cancer Data Set

We will now reuse the same cancer data set but use a different way to classify data into non-malignant and malignant categories: Support Vector Machines (SVM). SVMs are linear classifiers which means that they work best when data is linearly separable. In the case of the cancer data, there are nine dimensions of values that (hopefully) predict one of the two output classes (or categories). If we think of the first 9 columns of data as defining a 9-dimensional space, then SVM will work well when a 8-dimensional hyperplane separates the samples into the two output classes (categories).

To make this simpler to visualize, if we just had two input columns, that defines a two-dimensional space, and if a straight line can separate most of the examples into the two output categories, then the data is linearly separable so SVM is a good technique to use. The SVM algorithm is effectively determining the parameters defining this one-dimensional line (or in the cancer data case, the 9-dimensional hyperspace).

What if data is not linearly separable? Then use the backpropagation neural network code in the chapter "Backpropagation Neural Networks" or the deep learning code in the chapter "Using Armed Bear Common Lisp With DeepLearning4j" to create a model.

SVM is very efficient so it often makes sense to first try SVM and if trained models are not accurate enough then use neural networks, including deep learning.

The following listing of file *clml_svm_classifier.lisp* shows how to read data, build a model and evaluate the model with different test data. In line 15 we use the function **clml.svm.mu:svm** that requires the type of kernel function to use, the training data, and testing data. Just for reference, we usually use Gaussian kernel functions for processing numeric data and linear kernel functions for handling text in natural language processing applications. Here we use a Gaussian kernel.

The function **cancer-data-svm-example-read-data** defined on line 40 differs from how we read and processed data earlier because we need to separate out the *positive* and *negative* training examples. The data is split in the lexically scoped function in lines 42-52. The last block of code in lines 54-82 is just top-level test code that gets executed when the file *clml_svm_classifier.lisp* is loaded.

```
;; note; run SBCL using: sbcl --dynamic-space-size 2560

(ql:quickload '(:clml
                :clml.hjs ; utilities
                :clml.svm))

(defpackage #:clml-svm-classifier-example1
  (:use #:cl #:clml.hjs.read-data))

(in-package #:clml-svm-classifier-example1)

(defun svm-classifier-test (kernel train test)
  "train and test are lists of lists, with first elements being negative
   samples and the second elements being positive samples"
  (let ((decision-function (clml.svm.mu:svm kernel (cadr train) (car train)))
        (correct-positives 0)
        (wrong-positives 0)
        (correct-negatives 0)
        (wrong-negatives 0))
    ;; type: #<CLOSURE (LAMBDA (CLML.SVM.MU::Z) :IN CLML.SVM.MU::DECISION)>
    (print decision-function)
    (princ "***** NEGATIVE TESTS: calling decision function:")
    (terpri)
    (dolist (neg (car test))  ;; negative test examples
      (let ((prediction (funcall decision-function neg)))
        (print prediction)
        (if prediction (incf wrong-negatives) (incf correct-negatives))))
    (princ "***** POSITIVE TESTS: calling decision function:")
    (terpri)
    (dolist (pos (cadr test)) ;; positive test examples
      (let ((prediction (funcall decision-function pos)))
        (print prediction)
        (if prediction (incf correct-positives) (incf wrong-positives))))
    (format t "Number of correct negatives ~a~%" correct-negatives)
    (format t "Number of wrong negatives ~a~%" wrong-negatives)
    (format t "Number of correct positives ~a~%" correct-positives)
    (format t "Number of wrong positives ~a~%" wrong-positives)))


(defun cancer-data-svm-example-read-data ()

  (defun split-positive-negative-cases (data)
    (let ((negative-cases '())
```

```
          (positive-cases '()))
      (dolist (d data)
        ;;(print (list "*  d=" d))
        (if (equal (symbol-name (first (last d))) "benign")
            (setf negative-cases
                  (cons (reverse (cdr (reverse d))) negative-cases))
            (setf positive-cases
                  (cons (reverse (cdr (reverse d))) positive-cases))))
      (list negative-cases positive-cases)))

  (let* ((train1
          (clml.hjs.read-data:read-data-from-file
           "./machine_learning_data/labeled_cancer_training_data.csv"
           :type :csv
           :csv-type-spec (append (make-list 9 :initial-element 'double-float)
                                  '(symbol))))
         (train-as-list
          (split-positive-negative-cases
           (coerce
            (map 'list
                 #'(lambda (x) (coerce x 'list))
                 (coerce (clml.hjs.read-data:dataset-points train1) 'list))
            'list)))
         (test1
          (clml.hjs.read-data:read-data-from-file
           "./machine_learning_data/labeled_cancer_test_data.csv"
           :type :csv
           :csv-type-spec (append (make-list 9 :initial-element 'double-float)
                                  '(symbol))))
         (test-as-list
          (split-positive-negative-cases
           (coerce
            (map 'list
                 #'(lambda (x) (coerce x 'list))
                 (coerce (clml.hjs.read-data:dataset-points test1) 'list))
            'list))))

    ;; we will use a gaussian kernel for numeric data.
    ;; note: for text classification, use a clml.svm.mu:+linear-kernel+
    (svm-classifier-test
     (clml.svm.mu:gaussian-kernel 2.0d0)
     train-as-list test-as-list)))

```

```
87 (cancer-data-svm-example-read-data)
```

The sample code prints the prediction values for the test data which I will not show here. Here are the last four lines of output showing the cumulative statistics for the test data:

```
1 Number of correct negatives 219
2 Number of wrong negatives 4
3 Number of correct positives 116
4 Number of wrong positives 6
```

## CLML Wrap Up

The CLML machine learning library is under fairly active development and I showed you enough to get started: understanding the data APIs and examples for KNN clustering and SVM classification.

A good alternative to CLML is MGL[63] that supports backpropagation neural networks, boltzmann machines, and gaussian processes.

In the next two chapters we continue with the topic of machine learning with backpropagation andf Hopfield neural networks.

[63] https://github.com/melisgl/mgl

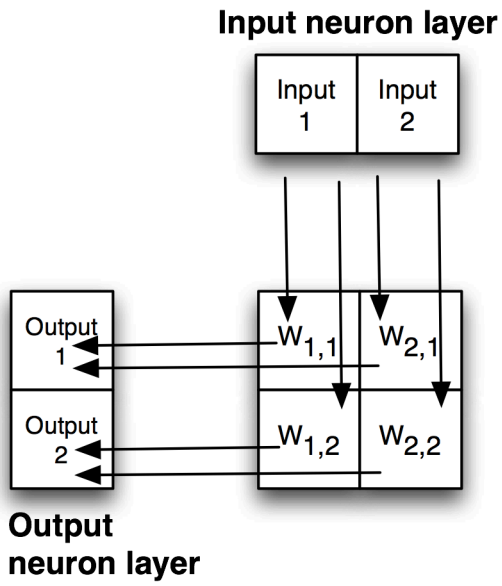
# Backpropagation Neural Networks

Let's start with an overview of how these networks work and then fill in more detail later. Backpropagation networks are trained by applying training inputs to the network input layer, propagate values through the network to the output neurons, compare the errors (or differences) between these propagated output values and the training data output values. These output errors are backpropagated though the network and the magnitude of backpropagated errors are used to adjust the weights in the network.

The example we look at here uses the **plotlib** package from an earlier chapter and the source code for the example is the file **loving_snippet/backprop_neural_network.lisp**.

We will use the following diagram to make this process more clear. There are four weights in this very simple network:

- $W^{1,1}$ is the floating point number representing the connection strength between input_neuron$^1$ and output_neuron$^1$
- $W^{2,1}$ connects input_neuron$^2$ to output_neuron$^1$
- $W^{1,2}$ connects input_neuron$^1$ to output_neuron$^2$
- $W^{2,2}$ connects input_neuron$^2$ to output_neuron$^2$




**Understanding how connection weights connect neurons in adjacent layers**

Before any training the weight values are all small random numbers.

Consider a training data element where the input neurons have values [0.1, 0.9] and the desired output neuron values are [0.9 and 0.1], that is flipping the input values. If the propagated output values for the current weights are [0.85, 0.5] then the value of the first output neuron has a small error abs(0.85 - 0.9) which is 0.05. However the propagated error of the second output neuron is high: abs(0.5 - 0.1) which is 0.4. Informally we see that the weights feeding input output neuron 1 ($W^{1,1}$ and $W^{2,1}$) don't need to be changed much but the neuron that feeding input neuron 2 ($W^{1,2}$ and $W^{2,2}$) needs modification (the value of $W^{2,2}$ is too large).

Of course, we would never try to manually train a network like this but it is important to have at least an informal understanding of how weights connect the flow of value (we will call this activation value later) between neurons.

In this neural network see in the first figure we have four weights connecting the input and output neurons. Think of these four weights forming a four-dimensional space where the range in each dimension is constrained to small positive and negative floating point values. At any point in this "weight space", the numeric values of the weights defines a model that maps the inputs to the outputs. The error seen at the output neurons is accumulated for each training example (applied to the input neurons). The training process is finding a point in this four-dimensional space that has low errors summed across the training data. We will use gradient descent to start with a random point in the four-dimensional space (i.e., an initial random set of weights) and move the point towards a local minimum that represents the weights in a model that is (hopefully) "good enough" at representing the training data.

This process is simple enough but there are a few practical considerations:

- Sometimes the accumulated error at a local minimum is too large even after many training cycles and it is best to just restart the training process with new random weights.
- If we don't have enough training data then the network may have enough memory capacity to memorize the training examples. This is not what we want: we want a model with just enough memory capacity (as represented by the number of weights) to form a generalized predictive model, but not so specific that it just memorizes the training examples. The solution is to start with small networks (few hidden neurons) and increase the number of neurons until the training data can be learned. In general, having a lot of training data is good and it is also good to use as small a network as possible.

In practice using backpropagation networks is an iterative process of experimenting with the size of a network.

In the example program (in the file *backprop_neural_network.lisp*) we use the plotting library developed earlier to visualize neuron activation and connecting weight values while the network trains.

The following three screen shots from running the function **test3** defined at the bottom of the file *backprop_neural_network.lisp* illustrate the process of starting with random weights, getting

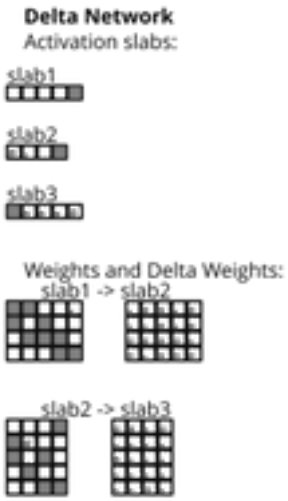
random outputs during initial training, and as delta weights are used to adjust the weights in a network, then the training examples are learned:


**At the start of the training run with random weights and large delta weights**

In the last figure the initial weights are random so we get random mid-range values at the output neurons.


**The trained weights start to produce non-random output**

As we start to train the network, adjusting the weights, we start to see variation in the output neurons as a function of what the inputs are.

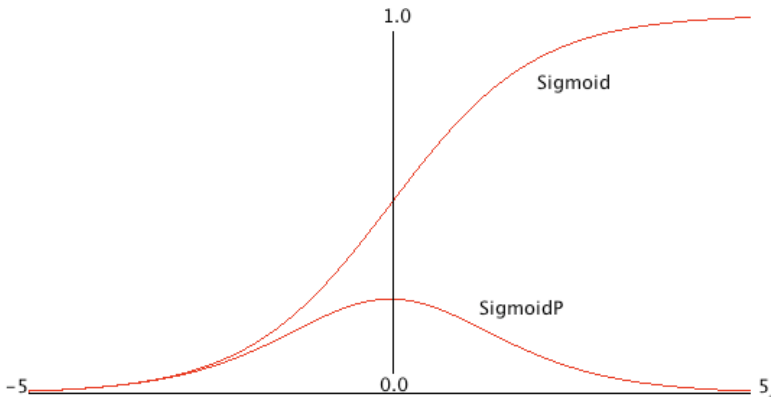



**After training many cycles the training examples are learned, with only small output errors**

In the last figure the network is trained sufficiently well to map inputs [0, 0, 0, 1] to output values that are approximately [0.8, 0.2, 0.2, 0.3] which is close to the expected value [1, 0, 0, 0].

The example source file *backprop_neural_network.lisp* is long so we will only look at the more interesting parts here. Specifically we will not look at the code to plot neural networks using **plotlib**.

The activation values of individual neurons are limited to the range [0, 1] by first calculating their values based on the sum activation values of neurons in the previous layer times the values of the connecting weights and then using the **Sigmoid** function to map the sums to the desired range. The Sigmoid function and the derivative of the **Sigmoid** function (**dSigmoid**) look like:




**Sigmoid and Derivative of the Sigmid Functions**

Here are the definitions of these functions:

```
(defun Sigmoid (x)
  (/ 1.0 (+ 1.0 (exp (- x)))))

(defun dSigmoid (x)
  (let ((temp (Sigmoid x)))
    (* temp (- 1.0 temp))))
```

The function **NewDeltaNetwork** creates a new neual network object. This code allocates storage for input, hidden, output layers (I sometimes refer to neuron layers as "slabs"), and the connection weights. Connection weights are initialized to small random values.

```
; (NewDeltaNetwork sizeList)
;       Args:   sizeList = list of sizes of slabs. This also defines
;                          the number of slabs in the network.
;                          (e.g.,  '(10 5 4) ==> a 3-slab network with 10
;                           input neurons, 5 hidden neurons, and 4 output
;                           neurons).
;
;       Returned value = a list describing the network:
;          (nLayers sizeList
;           (activation-array[1] .. activation-array[nLayers])
;           (weight-array[2] .. weight-array[nLayers])
;           (sum-of-products[2] .. sum-of-products[nLayers[nLayers])
;           (back-prop-error[2] .. back-prop-error[nLayers]))
;           (old-delta-weights[2] .. for momentum term

                                       :initial-element 0.0))
           (reverse old-dw-list)))

    ;;
     ;  Initialize values for all activations:
     ;;
    (mapc
     (lambda (x)
        (let ((num (array-dimension x 0)))
          (dotimes (n num)
            (setf (aref x n) (frandom 0.01 0.1)))))
     a-list)

    ;;
     ;  Initialize values for all weights:
     ;;
    (mapc
```

```
33      (lambda (x)
34         (let ((numI (array-dimension x 0))
35               (numJ (array-dimension x 1)))
36           (dotimes (j numJ)
37             (dotimes (i numI)
38               (setf (aref x i j) (frandom -0.5 0.5))))))
39       w-list)
40      (list numLayers sizeList a-list s-list w-list dw-list
41            d-list old-dw-list alpha beta)))
```

In the following listing the function **DeltaLearn** processes one pass through all of the training data. Function **DeltaLearn** is called repeatedly until the return value is below a desired error threshold. The main loop over each training example is implemented in lines 69-187. Inside this outer loop there are two phases of training for each training example: a forward pass propagating activation from the input neurons to the output neurons via any hidden layers (lines 87-143) and then the weight correcting backpropagation of output errors while making small adjustments to weights (lines 148-187):

```
1  ;;
2   ;  Utility function for training a delta rule neural network.
3   ;  The first argument is the name of an output PNG plot file
4   ;  and a nil value turns off plotting the network during training.
5   ;  The second argument is a network definition (as returned from
6   ;  NewDeltaNetwork), the third argument is a list of training
7   ;  data cases (see the example test functions at the end of this
8   ;  file for examples.
9   ;;
10
11 (defun DeltaLearn (plot-output-file-name
12                     netList trainList)
13   (let ((nLayers (car netList))
14         (sizeList (cadr netList))
15         (activationList (caddr netList))
16         (sumOfProductsList (car (cdddr netList)))
17         (weightList (cadr (cdddr netList)))
18         (deltaWeightList (caddr (cdddr netList)))
19         (deltaList (cadddr (cdddr netList)))
20         (oldDeltaWeightList (cadddr (cdddr (cdr netList))))
21         (alpha (cadddr (cdddr (cddr netList))))
22         (beta (cadddr (cdddr (cdddr netList))))
23         (inputs nil)
24         (targetOutputs nil)
25         (iDimension nil)
```

```
        (jDimension nil)
        (iActivationVector nil)
        (jActivationVector nil)
        (n nil)
        (weightArray nil)
        (sumOfProductsArray nil)
        (iDeltaVector nil)
        (jDeltaVector nil)
        (deltaWeightArray nil)
        (oldDeltaWeightArray nil)
        (sum nil)
        (iSumOfProductsArray nil)
        (error nil)
        (outputError 0)
        (delta nil)
        (eida nil)
        (inputNoise 0))

    ;;
     ; Zero out deltas:
     ;;
    (dotimes (n (- nLayers 1))
      (let* ((dw (nth n deltaList))
             (len1 (array-dimension dw 0)))
        (dotimes (i len1)
          (setf (aref dw i) 0.0))))

    ;;
     ; Zero out delta weights:
     ;;
    (dotimes (n (- nLayers 1))
      (let* ((dw (nth n deltaWeightList))
             (len1 (array-dimension dw 0))
             (len2 (array-dimension dw 1)))
        (dotimes (i len1)
          (dotimes (j len2)
            (setf (aref dw i j) 0.0)))))

    (setq inputNoise *delta-default-input-noise-value*)

    ;;
     ;  Main loop on training examples:
     ;;
```

```
69      (dolist (tl trainList)
70
71        (setq inputs (car tl))
72        (setq targetOutputs (cadr tl))
73
74        (if *delta-rule-debug-flag*
75          (print (list "Current targets:" targetOutputs)))
76
77        (setq iDimension (car sizeList)) ; get the size of the input slab
78        (setq iActivationVector (car activationList)) ; input activations
79        (dotimes (i iDimension) ; copy training inputs to input slab
80          (setf
81           (aref iActivationVector i)
82           (+ (nth i inputs) (frandom (- inputNoise) inputNoise))))
83        ;;
84         ; Propagate activation through all of the slabs:
85         ;;
86        (dotimes (n-1 (- nLayers 1))  ; update layer i to layer flowing to layer j
87          (setq n (+ n-1 1))
88          (setq jDimension (nth n sizeList)) ; get the size of the j'th layer
89          (setq jActivationVector (nth n activationList)) ; activation  for slab j
90          (setq weightArray (nth n-1 weightList))
91          (setq sumOfProductsArray (nth n-1 sumOfProductsList))
92          (dotimes (j jDimension) ; process each neuron in slab j
93            (setq sum 0.0) ; init sum of products to zero
94            (dotimes (i iDimension) ; activation from neurons in previous slab
95              (setq
96               sum
97               (+ sum (* (aref weightArray i j) (aref iActivationVector i)))))
98            (setf (aref sumOfProductsArray j) sum) ; save sum of products
99            (setf (aref jActivationVector j) (Sigmoid sum)))
100         (setq iDimension jDimension)     ; reset index for next slab pair
101         (setq iActivationVector jActivationVector))
102       ;;
103        ; Activation is  spread through the network and sum of products
104        ; calculated. Now modify the weights in the network using back
105        ; error propagation. Start by calculating the error signal for
106        ; each neuron in the output layer:
107        ;;
108       (setq jDimension (nth (- nLayers 1) sizeList)) ; size of last layer
109       (setq jActivationVector (nth (- nLayers 1) activationList))
110       (setq jDeltaVector (nth (- nLayers 2) deltaList))
111       (setq sumOfProductsArray (nth (- nLayers 2) sumOfProductsList))
```

```
112         (setq outputError 0)
113         (dotimes (j jDimension)
114           (setq delta (- (nth j targetOutputs) (aref jActivationVector j)))
115           (setq outputError (+ outputError (abs delta)))
116           (setf
117            (aref jDeltaVector j)
118            (+
119             (aref jDeltaVector j)
120             (* delta (dSigmoid (aref sumOfProductsArray j))))))
121         ;;
122          ; Now calculate the backpropagated error signal for all hidden slabs:
123          ;;
124         (dotimes (nn (- nLayers 2))
125           (setq n (- nLayers 3 nn))
126           (setq iDimension (nth (+ n 1) sizeList))
127           (setq iSumOfProductsArray (nth n sumOfProductsList))
128           (setq iDeltaVector (nth n deltaList))
129           (dotimes (i iDimension)
130             (setf (aref iDeltaVector i) 0.0))
131           (setq weightArray (nth (+ n 1) weightList))
132           (dotimes (i iDimension)
133             (setq error 0.0)
134             (dotimes (j jDimension)
135               (setq error
136                     (+ error (* (aref jDeltaVector j) (aref weightArray i j)))))
137             (setf
138              (aref iDeltaVector i)
139              (+
140               (aref iDeltaVector i)
141               (* error (dSigmoid (aref iSumOfProductsArray i))))))
142           (setq jDimension iDimension)
143           (setq jDeltaVector iDeltaVector))
144
145         ;;
146          ; Update all delta weights in the network:
147          ;;
148         (setq iDimension (car sizeList))
149         (dotimes (n (- nLayers 1))
150           (setq iActivationVector (nth n activationList))
151           (setq jDimension (nth (+ n 1) sizeList))
152           (setq jDeltaVector (nth n deltaList))
153           (setq deltaWeightArray (nth n deltaWeightList))
154           (setq weightArray (nth n weightList))
```

```
155         (setq eida (nth n eidaList))
156
157         (dotimes (j jDimension)
158           (dotimes (i iDimension)
159             (setq delta (* eida (aref jDeltaVector j) (aref iActivationVector i)))
160             (setf
161              (aref DeltaWeightArray i j)
162              (+ (aref DeltaWeightArray i j) delta)))) ; delta weight changes
163
164         (setq iDimension jDimension))
165
166      ;;
167       ; Update all weights in the network:
168       ;;
169      (setq iDimension (car sizeList))
170      (dotimes (n (- nLayers 1))
171        (setq iActivationVector (nth n activationList))
172        (setq jDimension (nth (+ n 1) sizeList))
173        (setq jDeltaVector (nth n deltaList))
174        (setq deltaWeightArray (nth n deltaWeightList))
175        (setq oldDeltaWeightArray (nth n oldDeltaWeightList))
176        (setq weightArray (nth n weightList))
177        (dotimes (j jDimension)
178          (dotimes (i iDimension)
179            (setf
180             (aref weightArray i j)
181             (+ (aref weightArray i j)
182                (* alpha (aref deltaWeightArray i j))
183                (* beta  (aref oldDeltaWeightArray i j))))
184             (setf (aref oldDeltaWeightArray i j) ; save current delta weights
185                   (aref deltaWeightArray i j)))) ; ...for next momentum term.
186        (setq iDimension jDimension))
187
188      (if plot-output-file-name
189          (DeltaPlot netList plot-output-file-name)))
190
191      (/ outputError jDimension)))
```

The function **DeltaRecall** in the next listing can be used with a trained network to calculate outputs for new input values:

```
;;
 ;  Utility for using a trained neural network in the recall mode.
 ;  The first argument to this function is a network definition (as
 ;  returned from NewDeltaNetwork) and the second argument is a list
 ;  of input neuron activation values to drive through the network.
 ;  The output is a list of the calculated activation energy for
 ;  each output neuron.
 ;;
(defun DeltaRecall (netList inputs)
  (let ((nLayers (car netList))
        (sizeList (cadr netList))
        (activationList (caddr netList))
        (weightList (cadr (cdddr netList)))
        (iDimension nil)
        (jDimension nil)
        (iActivationVector nil)
        (jActivationVector nil)
        (n nil)
        (weightArray nil)
        (returnList nil)
        (sum nil))
    (setq iDimension (car sizeList)) ; get the size of the input slab
    (setq iActivationVector (car activationList)) ; get input activations
    (dotimes (i iDimension) ; copy training inputs to input slab
      (setf (aref iActivationVector i) (nth i inputs)))
    (dotimes (n-1 (- nLayers 1))  ; update layer j to layer i
      (setq n (+ n-1 1))
      (setq jDimension (nth n sizeList)) ; get the size of the j'th layer
      (setq jActivationVector (nth n activationList)) ; activation for slab j
      (setq weightArray (nth n-1 weightList))
      (dotimes (j jDimension) ; process each neuron in slab j
        (setq sum 0.0) ; init sum of products to zero
        (dotimes (i iDimension) ; get activation from each neuron in last slab
          (setq
           sum
           (+ sum (* (aref weightArray i j) (aref iActivationVector i)))))
        (if *delta-rule-debug-flag*
          (print (list "sum=" sum)))
        (setf (aref jActivationVector j) (Sigmoid sum)))
      (setq iDimension jDimension) ; get ready for next slab pair
      (setq iActivationVector jActivationVector))
      (dotimes (j jDimension)
        (setq returnList (append returnList (list (aref jActivationVector j)))))
```

```
44        returnList))
```

We saw three output plots earlier that were produced during a training run using the following code:

```
1  (defun test3 (&optional (restart 'yes) &aux RMSerror) ; three layer network
2    (if
3      (equal restart 'yes)
4      (setq temp (newdeltanetwork '(5 4 5))))
5    (dotimes (ii 3000)
6      (let ((file-name
7       (if (equal (mod ii 400) 0)
8           (concatenate 'string "output_plot_" (format nil "~12,'0d" ii) ".png")
9           nil)))
10       (setq
11        RMSerror
12        (deltalearn
13           file-name temp
14           '(((1 0 0 0 0) (0 1 0 0 0))
15             ((0 1 0 0 0) (0 0 1 0 0))
16             ((0 0 1 0 0) (0 0 0 1 0))
17             ((0 0 0 1 0) (0 0 0 0 1))
18             ((0 0 0 0 1) (1 0 0 0 0)))))
19       (if (equal (mod ii 50) 0) ;; print error out every 50 cycles
20         (progn
21           (princ "....training cycle \#")
22           (princ ii)
23           (princ " RMS error = ")
24           (princ RMSerror)
25           (terpri))))))
```

Here the function **test3** defines training data for a very small test network for a moderately difficult function to learn: to rotate the values in the input neurons to the right, wrapping around to the first neuron. The start of the main loop in line calls the training function 3000 times, creating a plot of the network every 400 times through the main loop.

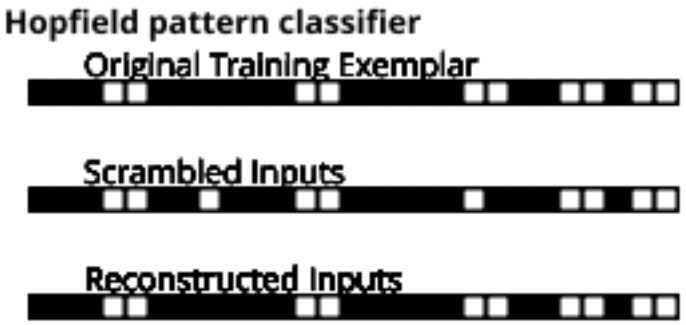
Backpropagation networks have been used sucessfully in production for about 25 years. In the next chapter we will look at a less practical type of network, Hopfield networks, that are still interesting because the in some sense Hopfield networks model how our brains work. In the final chapter we will look at deep learning neural networks.

# Hopfield Neural Networks

A Hopfield network[64] (named after John Hopfield) is a recurrent network since the flow of activation through the network has loops. These networks are trained by applying input patterns and letting the network settle in a state that stores the input patterns.

The example code is in the file **src/loving_snippets/Hopfield_neural_network.lisp**.

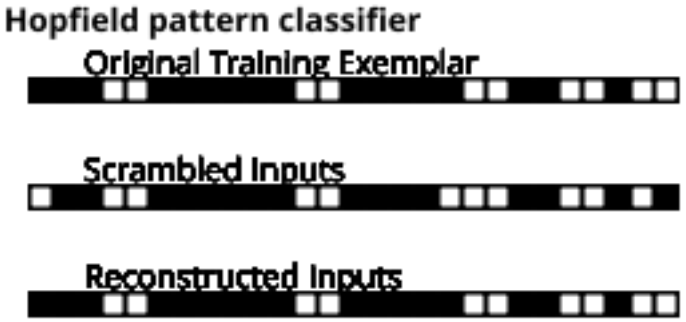
The example we look at recognizes patterns that are similar to the patterns seen in training examples and maps input patterns to a similar training input pattern. The following figure shows output from the example program showing an original training pattern, a similar pattern with one cell turned on and other off, and the reconstructed pattern:



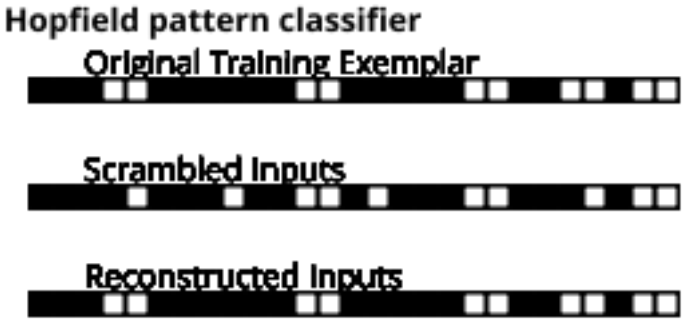

To be clear, we have taken one of the original input patterns the network has learned, slightly altered it, and applied it as input to the network. After cycling the network, the slightly scrambled input pattern we just applied will be used as an associative memory key, look up the original pattern, and rewrite to input values with the original learned pattern. These Hopfield networks are very different than backpropagation networks: neuron activation are forced to values of -1 or +1 and not be differentiable and there are no separate output neurons.

The next example has the values of three cells modified from the original and the original pattern is still reconstructed correctly:

[64] https://en.wikipedia.org/wiki/Hopfield_network




This last example has four of the original cells modified:




The following example program shows a type of content-addressable memory. After a Hopfield network learns a set of input patterns then it can reconstruct the original paterns when shown similar patterns. This reconstruction is not always perfecrt.

The following function **Hopfield-Init** (in file *Hopfield_neural_network.lisp*) is passed a list of lists of training examples that will be remembered in the network. This function returns a list containing the data defining a Hopfield neural network. All data for the network is encapsulated in the list returned by this function, so multiple Hopfield neural networks can be used in an application program.

In lines 9-12 we allocate global arrays for data storage and in lines 14-18 the training data is copied.

The inner function **adjustInput** on lines 20-29 adjusts data values to values of -1.0 or +1.0. In lines 31-33 we are initializing all of the weights in the Hopfield network to zero.

The last nested loop, on lines 35-52, calculates the autocorrelation weight matrix from the input test patterns.

On lines 54-56, the function returns a representation of the Hopfield network that will be used later in the function **HopfieldNetRecall** to find the most similar "remembered" pattern given a new (fresh) input pattern.

```
(defun Hopfield-Init (training-data
                      &aux temp *num-inputs* *num-training-examples*
                           *training-list* *inputCells* *tempStorage*
                           *HopfieldWeights*)

  (setq *num-inputs* (length (car training-data)))
  (setq *num-training-examples* (length training-data))

  (setq *training-list* (make-array (list *num-training-examples* *num-inputs*)))
  (setq *inputCells* (make-array (list *num-inputs*)))
  (setq *tempStorage* (make-array (list *num-inputs*)))
  (setq *HopfieldWeights* (make-array (list *num-inputs* *num-inputs*)))

  (dotimes (j *num-training-examples*) ;; copy training data
    (dotimes (i *num-inputs*)
      (setf
       (aref *training-list* j i)
       (nth i (nth j training-data)))))

  (defun adjustInput (value)  ;; this function is lexically scoped
    (if (< value 0.1)
      -1.0
      +1.0))

  (dotimes (i *num-inputs*) ;; adjust training data
    (dotimes (n *num-training-examples*)
      (setf
       (aref *training-list* n i)
       (adjustInput (aref *training-list* n i)))))

  (dotimes (i *num-inputs*) ;; zero weights
    (dotimes (j *num-inputs*)
      (setf (aref *HopfieldWeights* i j) 0)))

  (dotimes (j-1 (- *num-inputs* 1)) ;; autocorrelation weight matrix
    (let ((j (+ j-1 1)))
      (dotimes (i j)
        (dotimes (s *num-training-examples*)
          (setq temp
                (truncate
                 (+
                  (*  ;; 2 if's truncate values to -1 or 1:
                   (adjustInput (aref *training-list* s i))
```

```
44                    (adjustInput (aref *training-list* s j)))
45                   (aref *HopfieldWeights* i j))))
46           (setf (aref *HopfieldWeights* i j) temp)
47           (setf (aref *HopfieldWeights* j i) temp)))))
48   (dotimes (i *num-inputs*)
49     (setf (aref *tempStorage* i) 0)
50     (dotimes (j i)
51       (setf (aref *tempStorage* i)
52             (+ (aref *tempStorage* i) (aref *HopfieldWeights* i j)))))
53
54   (list ;; return the value of the Hopfield network data object
55    *num-inputs* *num-training-examples* *training-list*
56    *inputCells* *tempStorage* *HopfieldWeights*))
```

The following function **HopfieldNetRecall** iterates the network to let it settle in a stable pattern which we hope will be the original training pattern most closely resembling the noisy test pattern.

The inner (lexically scoped) function **deltaEnergy** defined on lines 9-12 calculates a change in energy from old input values and the autocorrelation weight matrix. The main code uses the inner functions to iterate over the input cells, possibly modifying the cell at index **i** delta energy is greater than zero. Remember that the lexically scoped inner functions have access to the variables for the number of inputs, the number of training examples, the list of training examples, the input cell values, tempoary storage, and the Hopfield network weights.

```
1  (defun HopfieldNetRecall (aHopfieldNetwork numberOfIterations)
2    (let ((*num-inputs* (nth 0 aHopfieldNetwork))
3          (*num-training-examples*  (nth 1 aHopfieldNetwork))
4          (*training-list* (nth 2 aHopfieldNetwork))
5          (*inputCells* (nth 3 aHopfieldNetwork))
6          (*tempStorage* (nth 4 aHopfieldNetwork))
7          (*HopfieldWeights* (nth 5 aHopfieldNetwork)))
8
9      (defun deltaEnergy (row-index y &aux (temp 0.0))  ;; lexically scoped
10       (dotimes (j *num-inputs*)
11         (setq temp (+ temp (* (aref *HopfieldWeights* row-index j) (aref y j)))))
12       (- (* 2.0 temp) (aref *tempStorage* row-index)))
13
14     (dotimes (ii numberOfIterations) ;; main code
15       (dotimes (i *num-inputs*)
16         (setf (aref *inputCells* i)
17               (if (> (deltaEnergy i *inputCells*) 0)
18                   1
19                   0))))))
```

Function **test** in the next listing uses three different patterns for each test. Note that only the last pattern gets plotted to the output graphics PNG file for the purpose of producing figures for this chapter. If you want to produce plots of other patterns, edit just the third pattern defined on line AAAAA. The following plotting functions are inner lexically scoped so they have access to the data defined in the enclosing **let** expression in lines 16-21:

- **plotExemplar** - plots a vector of data
- **plot-original-inputCells** - plots the original input cells from training data
- **plot-inputCells** - plots the modified input cells (a few cells randomly flipped in value)
- **modifyInput** - scrambles training inputs

```
1  (defun test (&aux aHopfieldNetwork)
2    (let ((tdata '(  ;; sample sine wave data with different periods:
3                   (1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0)
4                   (0 1 1 0 0 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 1 1 0 0 1 0)
5                   (0 0 0 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 0 0 1 1 0 1 1)))
6          (width 300)
7          (height 180))
8      (vecto::with-canvas (:width width :height height)
9        (plotlib:plot-string-bold 10 (- height 14) "Hopfield pattern classifier")
10
11       ;; Set up network:
12       (print tdata)
13       (setq aHopfieldNetwork (Hopfield-Init tdata))
14
15       ;; lexically scoped variables are accesible by inner functions:
16       (let ((*num-inputs* (nth 0 aHopfieldNetwork))
17             (*num-training-examples*  (nth 1 aHopfieldNetwork))
18             (*training-list* (nth 2 aHopfieldNetwork))
19             (*inputCells* (nth 3 aHopfieldNetwork))
20             (*tempStorage* (nth 4 aHopfieldNetwork))
21             (*HopfieldWeights* (nth 5 aHopfieldNetwork)))
22
23         (defun plotExemplar (row &aux (dmin 0.0) (dmax 1.0) (x 20) (y 40))
24           (let ((YSize (array-dimension *training-list* 1)))
25             (plotlib:plot-string (+ x 20) (- height (- y 10))
26                                  "Original Training Exemplar")
27             (dotimes (j Ysize)
28               (plotlib:plot-fill-rect
29                (+ x (* j plot-size+1)) (- height y) plot-size plot-size
30                (truncate (*
31                           (/  (- (aref *training-list* row j) dmin)
```

```
                                (- dmax dmin))
                             5)))
              (plotlib:plot-frame-rect (+ x (* j plot-size+1))
                                       (- height y) plot-size plot-size))))

(defun plot-original-inputCells (&aux (dmin 0.0) (dmax 1.0) (x 20) (y 80))
  (let ((Xsize (array-dimension *inputCells* 0)))
    (plotlib:plot-string (+ x 20) (- height (- y 10)) "Scrambled Inputs")
    (dotimes (j Xsize)
      (plotlib:plot-fill-rect
       (+ x (* j plot-size+1)) (- height y) plot-size plot-size
       (truncate (*
                  (/  (- (aref *inputCells* j) dmin) (- dmax dmin))
                  5)))
      (plotlib:plot-frame-rect (+ x (* j plot-size+1))
                               (- height y) plot-size plot-size))))

(defun plot-inputCells (&aux (dmin 0.0) (dmax 1.0) (x 20) (y 120))
  (let ((Xsize (array-dimension *inputCells* 0)))
    (plotlib:plot-string (+ x 20) (- height (- y 10))
                         "Reconstructed Inputs")
    (dotimes (j Xsize)
      (plotlib:plot-fill-rect
       (+ x (* j plot-size+1)) (- height y) plot-size plot-size
       (truncate (* (/
                     (- (aref *inputCells* j) dmin)
                     (- dmax dmin))
                    5)))
      (plotlib:plot-frame-rect
         (+ x (* j plot-size+1)) (- height y) plot-size plot-size))))

(defun modifyInput (arrSize arr)  ;; modify input array for testing
  (dotimes (i arrSize)
    (if (< (random 50) 5)
        (if (> (aref arr i) 0)
            (setf (aref arr i) -1)
            (setf (aref arr i) 1)))))

;; Test network on training data that is randomly modified:

(dotimes (iter 10) ;; cycle 10 times and make 10 plots
  (dotimes (s *num-training-examples*)
    (dotimes (i *num-inputs*)
```

```
75              (setf (aref *inputCells* i) (aref *training-list* s i)))
76            (plotExemplar s)
77            (modifyInput *num-inputs* *inputCells*)
78            (plot-original-inputCells)
79            (dotimes (call-net 5) ;; iterate Hopfield net 5 times
80              (HopfieldNetRecall aHopfieldNetwork 1)  ;; calling with 1 iteration
81              (plot-inputCells)))
82
83          (vecto::save-png
84           (concatenate
85            'string
86            "output_plot_hopfield_nn_" (format nil "~5,'0d" iter) ".png")))))))
```

The plotting functions in lines 23-62 use the *plotlib* library to make the plots you saw earlier. The function **modifyInput** in lines 64-69 randomly flips the values of the input cells, taking an original pattern and slightly modifying it.

Hopfield neural networks, at least to some extent, seem to model some aspects of human brains in the sense that they can function as content-addressable (also called associative) memories. Ideally a partial input pattern from a remembered input can reconstruct the complete original pattern. Another interesting feature of Hopfield networks is that these memories really are stored in a distributed fashion: some of the weights can be randomly altered and patterns are still remembered, but with more recall errors.

# Using Python Deep Learning Models In Common Lisp With a Web Services Interface

In older editions of this book I had an example of using the Java DeepLearning4J deep learning library using Armed Bear Common Lisp, implemented in Java. I no longer use hybrid Java and Common Lisp applications in my own work and I decided to remove this example and replace it with two projects that use simple Python web services that act as wrappers for state of the art deep learning models with Common Lisp clients in the subdirectories:

- **src/spacy_web_client**: use the spaCy deep learning models for general NLP. I sometimes use my own pure Common Lisp NLP libraries we saw in earlier chapters and sometimes I use a Common Lisp client calling deep learning libraries like spaCy and TensorFlow.
- **src/coref_web_client**: coreference or anaphora resolution is the act of replacing pronouns in text with the original nouns that they refer to. This has traditionally been a very difficult and only partially solved problem until recent advances in deep learning models like BERT.

Note: in the next chapter we will cover similar functionality but we will use the **py4cl** library to more directly use Python and libraries like spaCy by starting another Python process and using streams for communication.

## Setting up the Python Web Services Used in this Chapter

You will need python and pip installed on your system. The source e code for the Python web services is found in the directory **loving-common-lisp/python**.

## Installing the spaCY NLP Services

I assume that you have some familiarity with using Python. If not, you will still be able to follow these directions assuming that you have the utilities **pip**, and **python** installed. I recommend installing Python and Pip using Anaconda[65].

[65] https://anaconda.org/anaconda/conda

The server code is in the subdirectory **python/python_spacy_nlp_server** where you will work when performing a one time initialization. After the server is installed you can then run it from the command line from any directory on your laptop.

I recommend that you use virtual Python environments when using Python applications to separate the dependencies required for each application or development project. Here I assume that you are running in a Python version 3.6 or higher environment. First you must install the dependencies:

```
pip install -U spacy
python -m spacy download en
pip install falcon
```

Then change directory to the subdirectory **python/python_spacy_nlp_server** in the git repo for this book and install the NLP server:

```
cd python/python_spacy_nlp_server
python setup.py install
```

Once you install the server, you can run it from any directory on your laptop or server using:

```
spacynlpserver
```

I use deep learning models written in Python using TensorFlow or PyTorch and provide Python web services that can be used in applications I write in Haskell or Common Lisp using web client interfaces for the services written in Python. While it is possible to directly embed models in Haskell and Common Lisp, I find it much easier and developer friendly to wrap deep learning models I use a REST services as I have done here. Often deep learning models only require about a gigabyte of memory and using pre-trained models has lightweight CPU resource needs so while I am developing on my laptop I might have two or three models running and available as wrapped REST services. For production, I configure both the Python services and my Haskell and Common Lisp applications to start automatically on system startup.

This is not a Python programming book and I will not discuss the simple Python wrapping code but if you are also a Python developer you can easily read and understand the code.

# Installing the Coreference NLP Services

I recommend that you use virtual Python environments when using Python applications to separate the dependencies required for each application or development project. Here I assume that you are running in a Python version 3.6 environment. First you should install the dependencies:

```
pip install spacy==2.1.0
pip install neuralcoref
pip install falcon
```

As I write this chapter the *neuralcoref* model and library require a slightly older version of SpaCy (the current latest version is 2.1.4).

Then change directory to the subdirectory **python/python_coreference_anaphora_resolution_-server** in the git repo for this book and install the coref server:

```
cd python_coreference_anaphora_resolution_server
python setup.py install
```

Once you install the server, you can run it from any directory on your laptop or server using:

```
corefserver
```

While. as we saw in the last example, it is possible to directly embed models in Haskell and Common Lisp, I find it much easier and developer friendly to wrap deep learning models I use a REST services as I have done here. Often deep learning models only require about a gigabyte of memory and using pre-trained models has lightweight CPU resource needs so while I am developing on my laptop I might have two or three models running and available as wrapped REST services. For production, I configure both the Python services and my Haskell and Common Lisp applications to start automatically on system startup.

This is not a Python programming book and I will not discuss the simple Python wrapping code but if you are also a Python developer you can easily read and understand the code.

## Common Lisp Client for the spaCy NLP Web Services

Before looking at the code, I will show you typical output from running this example:

```
$ sbcl
This is SBCL 1.3.16, an implementation of ANSI Common Lisp.
* (ql:quickload "spacy-web-client")
To load "spacy":
  Load 1 ASDF system:
    spacy-web-client
; Loading "spacy-web-client"
.........
("spacy-web-client")
* (defvar x
```

```
   (spacy-web-client:spacy-client
  "President Bill Clinton went to Congress. He gave a speech on taxes and Mexico."))
* (spacy-web-client:spacy-data-entities x)
"Bill Clinton/PERSON"
* (spacy-web-client:spacy-data-tokens x)
("President" "Bill" "Clinton" "went" "to" "Congress" "." "He" "gave" "a"
 "speech" "on" "taxes" "and" "Mexico" ".")
```

The client library is implemented in the file **src/spacy_web_client/spacy-web-client.lisp**:

```
1  (in-package spacy-web-client)
2
3  (defvar base-url "http://127.0.0.1:8008?text=")
4
5  (defstruct spacy-data entities tokens)
6
7  (defun spacy-client (query)
8    (let* ((the-bytes
9            (drakma:http-request
10            (concatenate 'string
11                         base-url
12                         (do-urlencode:urlencode  query))
13            :content-type "application/text"))
14          (fetched-data
15           (flexi-streams:octets-to-string the-bytes :external-format :utf-8))
16          (lists (with-input-from-string (s fetched-data)
17                   (json:decode-json s))))
18     (print lists)
19     (make-spacy-data :entities (cadar lists) :tokens (cdadr lists))))
```

On line 3 we define base URL for accessing the spaCy web service, assuming that it is running on your laptop and not a remote server. On line 5 we define a **defstruct** named **spacy-data** that has two fields: a list of entities in the input text and a list of word tokens in the input text.

The function **spacy-client** builds a query string on lines 10-12 that consists of the **base-url** and the input query text URL encoded. The drakma library, that we used before, is used to make a HTTP request from the Python spaCy server. Lines 14-15 uses the flexi-streams package to convert raw byte data to UTF8 characters. Lines 16-17 use the json package to parse the UTF8 encoded string, getting two lists of strings. I left the debug printout expression in line 18 so that you can see the results of parsing the JSON data. The function **make-spacy-data** was generated for us by the **defstruct** statement on line 5.

# Common Lisp Client for the Coreference NLP Web Services

Let's look at some typical output from this example, then we will look at the code:

```
$ sbcl
This is SBCL 1.3.16, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.

#P"/Users/markw/quicklisp/setup.lisp"
"starting up quicklisp"
* (ql:quickload "coref")
To load "coref":
  Load 1 ASDF system:
    coref
; Loading "coref"
.................................................
[package coref]
("coref")
* (coref:coref-client "My sister has a dog Henry. She loves him.")

"My sister has a dog Henry. My sister loves a dog Henry."
* (coref:coref-client "My sister has a dog Henry. He often runs to her.")

"My sister has a dog Henry. a dog Henry often runs to My sister."
```

Notice that pronouns in the input text are correctly replaced by the noun phrases that the pronoun refer to.

The implementation for the core client is in the file **src/coref_web_client/coref.lisp**:

```
(in-package #:coref)

;; (ql:quickload :do-urlencode)

(defvar base-url "http://127.0.0.1:8000?text=")

(defun coref-client (query)
  (let ((the-bytes
         (drakma:http-request
          (concatenate 'string
                       base-url
                       (do-urlencode:urlencode  query)
                       "&no_detail=1")
          :content-type "application/text")))
    (flexi-streams:octets-to-string the-bytes :external-format :utf-8)))
```

This code is similar to the example in the last section for setting up a call to **http-request** but is simpler: here the Python coreference web service accepts a string as input and returns a string as output with pronouns replaced by the nouns or noun phrases that they refer to. The example in the last section had to parse returned JSON data, this example does not.

## Trouble Shooting Possible Problems - Skip if this Example Works on Your System

If you run Common Lisp in an IDE (for example in LispWorks' IDE or VSCode with a Common Lisp plugin) make sure you start the IDE from the command line so your PATH environment variable will be set as it is in our bash or zsh shell.

Make sure you are starting your Common Lisp program or running a Common Lisp repl with the same Python installation (if you have Quicklisp installed, then you also have the package **uiop** installed):

```
$ which python
/Users/markw/bin/anaconda3/bin/python
$ sbcl
This is SBCL 2.0.2, an implementation of ANSI Common Lisp.
* (uiop:run-program "which python" :output :string)
"/Users/markw/bin/anaconda3/bin/python"
nil
0
*
```

## Python Interop Wrap-up

Much of my professional work in the last five years involved deep learning models and currently most available software is written in Python. While there are available libraries for calling Python code from Common Lisp, these libraries tend to not work well for Python code using libraries like TensorFlow, spaCy, PyTorch, etc., especially if the Python code is configured to use GPUs via CUDA of special hardware like TPUs. I find it simpler to simply wrap functionality implemented in Python as a simple web service.

# Using the PY4CL Library to Embed Python in Common Lisp

We will tackle the same problem as the previous chapter but take a different approach. Now we will use Ben Dudson's project Py4CL[66] that automatically starts a Python process and communicates with the Python process via a stream interface. The approach we took before is appropriate for large scale systems where you might want scale horizontally by having Python processes running on different servers than the servers used for the Common Lisp parts of your application. The approach we now take is much more convenient for what I call "laptop development" where the management of a Python process and communication is handled for you by the Py4CL library. If you need to build multi-server distributed systems for scaling reasons then use the examples in the last chapter.

While Py4CL provides a lot of flexibility for passing primitive types between Common Lisp and Python (in both directions), I find it easiest to write small Python wrappers that only use lists, arrays, numbers, and strings as arguments and return types. You might want to experiment with the examples on the Py4CL GitHub page that let you directly call Python libraries without writing wrappers. When I write code for my own projects I try to make code as simple as possible so when I need to later revisit my own code it is immediately obvious what it is doing. Since I have been using Common Lisp for almost 40 years, I often find myself reusing bits of my own old code and I optimize for making this as easy as possible. In other words I favor readability over "clever" code.

## Project Structure, Building the Python Wrapper, and Running an Example

The packaging of the Lisp code for my **spacy-py4cl** package is simple. Here is the listing of **package.lisp** for this project:

```
;;;; package.lisp

(defpackage #:spacy-py4cl
  (:use #:cl #:py4cl)
  (:export #:nlp))
```

Listing of **spacy-py4cl.asd**:

[66] https://github.com/bendudson/py4cl/

```
;;;; spacy-py4cl.asd

(asdf:defsystem #:spacy-py4cl
  :description "Use py4cl to use Python spaCy library embedded in Common Lisp"
  :author "Mark Watson <markw@markwatson.com>"
  :license "Apache 2"
  :depends-on (#:py4cl)
  :serial t
  :components ((:file "package")
               (:file "spacy-py4cl")))
```

You need to run a Python setup procedure to install the Python wrapper for space-py4cl on your system. Some output is removed for conciseness:

```
$ cd loving-common-lisp/src/spacy-py4cl
$ cd PYTHON_SPACY_SETUP_install/spacystub
$ pip install -U spacy
$ python -m spacy download en
$ python setup.py install
running install
running build
running build_py
running install_lib
running install_egg_info
Writing /Users/markw/bin/anaconda3/lib/python3.7/site-packages/spacystub-0.21-py3.7.\
egg-info
```

You only need to do this once unless you update to a later version of Python on your system.

If you are not familiar with Python, it is worth looking at the wrapper implementation, otherwise skip the next few paragraphs.

```
$ ls -R PYTHON_SPACY_SETUP_install
spacystub

PYTHON_SPACY_SETUP_install/spacystub:
README.md               setup.py        spacystub

PYTHON_SPACY_SETUP_install/spacystub/build/lib:
spacystub

PYTHON_SPACY_SETUP_install/spacystub/spacystub:
parse.py
```

Here is the implementation of **setup.py** that specifies how to build and install the wrapper globally for use on your system:

```
from distutils.core import setup

setup(name='spacystub',
      version='0.21',
      packages=['spacystub'],
      license='Apache 2',
      py_modules=['pystub'],
      long_description=open('README.md').read())
```

The definition of the library in file **PYTHON_SPACY_SETUP_install/spacystub/spacystub/parse.py**:

```
import spacy

nlp = spacy.load("en")

def parse(text):
  doc = nlp(text)
  response = {}
  response['entities'] = [(ent.text, ent.start_char, ent.end_char, ent.label_) for e\
nt in doc.ents]
  response['tokens'] = [token.text for token in doc]
  return [response['tokens'], response['entities']]
```

Here is a Common Lisp repl session showing you how to use the library implemented in the next section:

```
$ ccl
Clozure Common Lisp Version 1.12  DarwinX8664

For more information about CCL, please see http://ccl.clozure.com.

CCL is free software.  It is distributed under the terms of the Apache Licence, Vers\
ion 2.0.
? (ql:quickload "spacy-py4cl")
To load "spacy-py4cl":
  Load 1 ASDF system:
    spacy-py4cl
; Loading "spacy-py4cl"
[package spacy-py4cl]
```

```
("spacy-py4cl")
? (spacy-py4cl:nlp "The President of Mexico went to Canada")
#(#("The" "President" "of" "Mexico" "went" "to" "Canada") #(("Mexico" 17 23 "GPE") (\
"Canada" 32 38 "GPE")))
? (spacy-py4cl:nlp "Bill Clinton bought a red car. He drove it to the beach.")
#(#("Bill" "Clinton" "bought" "a" "red" "car" "." "He" "drove" "it" "to" "the" "beac\
h" ".") #(("Bill Clinton" 0 12 "PERSON")))
```

Entities in text are identified with the starting and ending character indices that refer to the input string. For example, the entity "Mexico" starts at character position 17 and character index 23 is the character after the entity name in the input string. The entity type "GPE" refers to a country name and "PERSON" refers to a person's name in the input text.

# Implementation of spacy-py4cl

The Common Lisp implementation for this package is simple. In line 5 the call to **py4cl:python-exec** starts a process to run Python and imports the function **parse** from my Python wrapper. The call to **py4cl:import-function** in line 6 finds a function named "parse" in the attached Python process and generates a Common Lisp function with the same name that handles calling into Python and converting handling the returned values to Common Lisp values:

```
;;;; spacy-py4cl.lisp

(in-package #:spacy-py4cl)

(py4cl:python-exec "from spacystub.parse import parse")
(py4cl:import-function "parse")

(defun nlp (text)
  (parse text))
```

While it is possible to call Python libraries directly using Py4CL, when I need to frequently use Python libraries like spaCY, TensorFlow, fast.ai, etc. in Common Lisp, I like to use wrappers that use simple as possible data types and APIs to communicate between a Common Lisp process and the spawned Python process.

# Trouble Shooting Possible Problems - Skip if this Example Works on Your System

When you install my wrapper library in Python on the command line whatever your shell if (bash, zsh, etc.) you should then try to import the library in a Python repl:

```
$ python
Python 3.7.4 (default, Aug 13 2019, 15:17:50)
[Clang 4.0.1 (tags/RELEASE_401/final)] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from spacystub.parse import parse
>>> parse("John Smith is a Democrat")
[['John', 'Smith', 'is', 'a', 'Democrat'], [('John Smith', 0, 10, 'PERSON'), ('Democ\
rat', 16, 24, 'NORP')]]
>>>
```

If this works and the Common Lisp library **spacy-py4cl** does not, then make sure you are starting your Common Lisp program or running a Common Lisp repl with the same Python installation (if you have Quicklisp installed, then you also have the package **uiop** installed):

```
$ which python
/Users/markw/bin/anaconda3/bin/python
$ sbcl
This is SBCL 2.0.2, an implementation of ANSI Common Lisp.
* (uiop:run-program "which python" :output :string)
"/Users/markw/bin/anaconda3/bin/python"
nil
0
*
```

If you run Common Lisp in an IDE (for example in LispWorks' IDE or VSCode with a Common Lisp plugin) make sure you start the IDE from the command line so your PATH environment variable will be set as it is in our bash or zsh shell.

## Wrap-up for Using Py4CL

While I prefer Common Lisp for general development and also AI research, there are useful Python libraries that I want to integrate into my projects. I hope that the last chapter and this chapter provide you with two solid approaches for you to use in your own work to take advantage of Python libraries.

# Semantic Web and Linked Data

I have written two previous books on the semantic web and linked data and most of my programming books have semantic web examples. Please note that the background material here on the semantic web standards RDF, RDFS, and SPARQL is shared with my book Practical Artificial Intelligence Programming With Java[67] so if you have read that book then the first several pages of this chapter will seem familiar.

Construction of Knowledge Graphs, as we will do in later chapters, is a core technology at many corporations and organizations to prevent data silos where different database systems are poorly connected and not as useful in combination than they could be. The use of RDF data stores is a powerful technique for data interoperability within organizations. Semantic Web standards like RDF, RDFS, and SPARQL support both building Knowledge Graphs and also key technologies for automating the collection and use of web data.

I worked as a contractor at Google on an internal Knowledge Graph project and I currently work at Olive AI[68] on their Knowledge Graph team.

The semantic web is intended to provide a massive linked set of data for use by software systems just as the World Wide Web provides a massive collection of linked web pages for human reading and browsing. The semantic web is like the web in that anyone can generate any content that they want. This freedom to publish anything works for the web because we use our ability to understand natural language to interpret what we read – and often to dismiss material that based upon our own knowledge we consider to be incorrect.

Semantic web and linked data technologies are also useful for smaller amounts of data, an example being a Knowledge Graph containing information for a business. We will further explore Knowledge Graphs in the next two chapters.

The core concept for the semantic web is data integration and use from different sources. As we will soon see, the tools for implementing the semantic web are designed for encoding data and sharing data from many different sources.

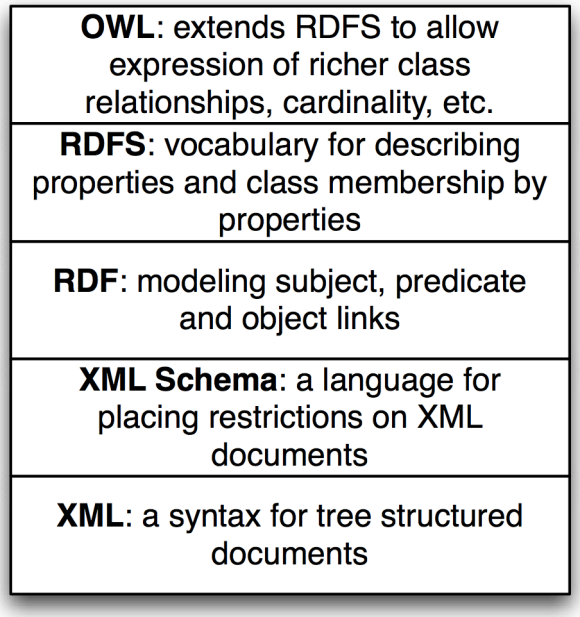
I cover the semantic web in this book because I believe that semantic web technologies are complementary to AI systems for gathering and processing data on the web. As more web pages are generated by applications (as opposed to simply showing static HTML files) it becomes easier to produce both HTML for human readers and semantic data for software agents.

There are several very good semantic web toolkits for the Java language and platform. Here we use Apache Jena because it is what I often use in my own work and I believe that it is a good starting technology for your first experiments with semantic web technologies. This chapter provides an

[67] https://leanpub.com/javaai
[68] https://oliveai.com

incomplete coverage of semantic web technologies and is intended as a gentle introduction to a few useful techniques and how to implement those techniques in Java. This chapter is the start of a journey in the technology that I think is as important as technologies like deep learning that get more public mindshare.

The following figure shows a layered hierarchy of data models that are used to implement semantic web applications. To design and implement these applications we need to think in terms of physical models (storage and access of RDF, RDFS, and perhaps OWL data), logical models (how we use RDF and RDFS to define relationships between data represented as unique URIs and string literals and how we logically combine data from different sources) and conceptual modeling (higher level knowledge representation and reasoning using OWL). Originally RDF data was serialized as XML data but other formats have become much more popular because they are easier to read and manually create. The top three layers in the figure might be represented as XML, or as LD-JSON (linked data JSON) or formats like N-Triples and N3 that we will use later.




**Semantic Web Data Models**

# Resource Description Framework (RDF) Data Model

The Resource Description Framework (RDF) is used to encode information and the RDF Schema (RDFS) facilitates using data with different RDF encodings without the need to convert one set of schemas to another. Later, using OWL we can simply declare that one predicate is the same as another, that is, one predicate is a sub-predicate of another (e.g., a property **containsCity** can be declared to be a sub-property of **containsPlace** so if something contains a city then it also contains a place), etc. The predicate part of an RDF statement often refers to a property.

RDF data was originally encoded as XML and intended for automated processing. In this chapter we will use two simple to read formats called "N-Triples" and "N3." Apache Jena can be used to convert between all RDF formats so we might as well use formats that are easier to read and understand. RDF data consists of a set of triple values:

- subject
- predicate
- object

Some of my work with semantic web technologies deals with processing news stories, extracting semantic information from the text, and storing it in RDF. I will use this application domain for the examples in this chapter and the next chapter when we implement code to automatically generate RDF for Knowledge Graphs. I deal with triples like:

- subject: a URL (or URI) of a news article.
- predicate: a relation like "containsPerson".
- object: a literal value like "Bill Clinton" or a URI representing Bill Clinton.

In the next chapter we will use the entity recognition library we developed in an earlier chapter to create RDF from text input.

We will use either URIs or string literals as values for objects. We will always use URIs for representing subjects and predicates. In any case URIs are usually preferred to string literals. We will see an example of this preferred use but first we need to learn the N-Triple and N3 RDF formats.

I proposed the idea that RDF was more flexible than Object Modeling in programming languages, relational databases, and XML with schemas. If we can tag new attributes on the fly to existing data, how do we prevent what I might call "data chaos" as we modify existing data sources? It turns out that the solution to this problem is also the solution for encoding real semantics (or meaning) with data: we usually use unique URIs for RDF subjects, predicates, and objects, and usually with a preference for not using string literals. The definitions of predicates are tied to a namespace and later with OWL we will state the equivalence of predicates in different namespaces with the same semantic meaning. I will try to make this idea more clear with some examples and Wikipedia has a good writeup on RDF[69].

Any part of a triple (subject, predicate, or object) is either a URI or a string literal. URIs encode namespaces. For example, the containsPerson predicate in the last example could be written as:

```
http://knowledgebooks.com/ontology/#containsPerson
```

The first part of this URI is considered to be the namespace for this predicate "containsPerson." When different RDF triples use this same predicate, this is some assurance to us that all users of this

[69] https://en.wikipedia.org/wiki/Resource_Description_Framework

predicate understand to the same meaning. Furthermore, we will see later that we can use RDFS to state equivalency between this predicate (in the namespace http://knowledgebooks.com/ontology/) with predicates represented by different URIs used in other data sources. In an "artificial intelligence" sense, software that we write does not understand predicates like "containsCity", "containsPerson", or "isLocation" in the way that a human reader can by combining understood common meanings for the words "contains", "city", "is", "person", and "location" but for many interesting and useful types of applications that is fine as long as the predicate is used consistently. We will see shortly that we can define abbreviation prefixes for namespaces which makes RDF and RDFS files shorter and easier to read.

The Jena library supports most serialization formats for RDF:

- Turtle
- N3
- N-Triples
- NQuads
- TriG
- JSON-LD
- RDF/XML
- RDF/JSON
- TriX
- RDF Binary

A statement in N-Triple format consists of three URIs (two URIs and a string literals for the object) followed by a period to end the statement. While statements are often written one per line in a source file they can be broken across lines; it is the ending period which marks the end of a statement. The standard file extension for N-Triple format files is *.nt and the standard format for N3 format files is *.n3.

My preference is to use N-Triple format files as output from programs that I write to save data as RDF. N-Triple files don't use any abbreviations and each RDF statement is self-contained. I often use tools like the command line commands in Jena or RDF4J to convert N-Triple files to N3 or other formats if I will be reading them or even hand editing them. Here is an example using the N3 syntax:

```
@prefix kb:  <http://knowledgebooks.com/ontology#>

<http://news.com/201234/> kb:containsCountry "China" .
```

The N3 format adds prefixes (abbreviations) to the N-Triple format. In practice it would be better to use the URI **http://dbpedia.org/resource/China** instead of the literal value "China."

Here we see the use of an abbreviation prefix "kb:" for the namespace for my company KnowledgeBooks.com ontologies. The first term in the RDF statement (the subject) is the URI of a news article. The second term (the predicate) is "containsCountry" in the "kb:" namespace. The last item in the

statement (the object) is a string literal "China." I would describe this RDF statement in English as, "The news article at URI http://news.com/201234 mentions the country China."

This was a very simple N3 example which we will expand to show additional features of the N3 notation. As another example, let's look at the case if this news article also mentions the USA. Instead of adding a whole new statement like this we can combine them using N3 notation. Here we have two separate RDF statements:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234/>
  kb:containsCountry
  <http://dbpedia.org/resource/China>  .

<http://news.com/201234/>
  kb:containsCountry
  <http://dbpedia.org/resource/United_States>  .
```

We can collapse multiple RDF statements that share the same subject and optionally the same predicate:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234/>
  kb:containsCountry
    <http://dbpedia.org/resource/China> ,
    <http://dbpedia.org/resource/United_States>  .
```

The indentation and placement on separate lines is arbitrary - use whatever style you like that is readable. We can also add in additional predicates that use the same subject (I am going to use string literals here instead of URIs for objects to make the following example more concise but in practice prefer using URIs):

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234/>
        kb:containsCountry "China" ,
                           "USA" .
        kb:containsOrganization "United Nations" ;
        kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                          "Hu Jintao" , "George W. Bush" ,
                          "Pervez Musharraf" ,
                          "Vladimir Putin" ,
                          "Mahmoud Ahmadinejad" .
```

This single N3 statement represents ten individual RDF triples. Each section defining triples with the same subject and predicate have objects separated by commas and ending with a period. Please note that whatever RDF storage system you use (we will be using Jena) it makes no difference if we load RDF as XML, N-Triple, of N3 format files: internally subject, predicate, and object triples are stored in the same way and are used in the same way. RDF triples in a data store represent directed graphs that may not all be connected.

I promised you that the data in RDF data stores was easy to extend. As an example, let us assume that we have written software that is able to read online news articles and create RDF data that captures some of the semantics in the articles. If we extend our program to also recognize dates when the articles are published, we can simply reprocess articles and for each article add a triple to our RDF data store using a form like:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .

<http://news.com/201234/> kb:datePublished "2008-05-11" .
```

Here we just represent the date as a string. We can add a type to the object representing a specific date:

```
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix kb:  <http://knowledgebooks.com/ontology#> .

 <http://news.com/201234/> kb:datePublished "2008-05-11"^^xsd:date .
```

Furthermore, if we do not have dates for all news articles that is often acceptable because when constructing SPARQL queries you can match optional patterns. If for example you are looking up articles on a specific subject then some results may have a publication date attached to the results for that article and some might not. In practice RDF supports types and we would use a date type as seen in the last example, not a string. However, in designing the example programs for this chapter I decided to simplify our representation of URIs and often use string literals as simple Java strings. For many applications this isn't a real limitation.

# Extending RDF with RDF Schema

RDF Schema (RDFS) supports the definition of classes and properties based on set inclusion. In RDFS classes and properties are orthogonal. Let's start with looking at an example using additional namespaces:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#>
@prefix dbo: <http://dbpedia.org/ontology/>

<http://news.com/201234/>
  kb:containsCountry
  <http://dbpedia.org/resource/China>   .

<http://news.com/201234/>
  kb:containsCountry
  <http://dbpedia.org/resource/United_States>   .

<http://dbpedia.org/resource/China>
  rdfs:label "China"@en,
  rdf:type dbo:Place ,
  rdf:type dbo:Country .
```

Because the semantic web is intended to be processed automatically by software systems it is encoded as RDF. There is a problem that must be solved in implementing and using the semantic web: everyone who publishes semantic web data is free to create their own RDF schemas for storing data; for example, there is usually no single standard RDF schema definition for topics like news stories and stock market data. The SKOS[70] is a namespace containing standard schemas and the most widely used standard is schema.org[71]. Understanding the ways of integrating different data sources using different schemas helps to understand the design decisions behind the semantic web applications. In this chapter I often use my own schemas in the knowledgebooks.com namespace for the simple examples you see here. When you build your own production systems part of the work is searching through **schema.org** and **SKOS** to use standard name spaces and schemas when possible. The use of standard schemas helps when you link internal proprietary Knowledge Graphs used in organization with public open data from sources like WikiData[72] and DBPedia[73].

We will start with an example that is an extension of the example in the last section that also uses RDFS. We add a few additional RDF statements:

[70] https://www.w3.org/2009/08/skos-reference/skos.html
[71] https://schema.org/docs/schemas.html
[72] https://www.wikidata.org/wiki/Wikidata:Main_Page
[73] https://wiki.dbpedia.org/about

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .
kb:containsCountry rdfs:subPropertyOf kb:containsPlace .
kb:containsState rdfs:subPropertyOf kb:containsPlace .
```

The last three lines declare that:

- The property containsCity is a sub-property of containsPlace.
- The property containsCountry is a sub-property of containsPlace.
- The property containsState is a sub-property of containsPlace.

Why is this useful? For at least two reasons:

- You can query an RDF data store for all triples that use property containsPlace and also match triples with properties equal to containsCity, containsCountry, or containsState. There may not even be any triples that explicitly use the property containsPlace.
- Consider a hypothetical case where you are using two different RDF data stores that use different properties for naming cities: **cityName** and **city**. You can define **cityName** to be a sub-property of **city** and then write all queries against the single property name **city**. This removes the necessity to convert data from different sources to use the same Schema. You can also use OWL to state property and class equivalency.

In addition to providing a vocabulary for describing properties and class membership by properties, RDFS is also used for logical inference to infer new triples, combine data from different RDF data sources, and to allow effective querying of RDF data stores. We will see examples of all of these features of RDFS when we later start using the Jena libraries to perform SPARQL queries.

# The SPARQL Query Language

SPARQL is a query language used to query RDF data stores. While SPARQL may initially look like SQL, we will see that there are some important differences like support for RDFS and OWL inferencing and graph-based instead of relational matching operations. We will cover the basics of SPARQL in this section and then see more examples later when we learn how to embed Jena in Java applications, and see more examples in the last chapter Knowledge Graph Navigator.

We will use the N3 format RDF file test_data/news.n3 for the examples. I created this file automatically by spidering Reuters news stories on the news.yahoo.com web site and automatically extracting named entities from the text of the articles. We saw techniques for extracting named entities from text in earlier chapters. In this chapter we use these sample RDF files.

You have already seen snippets of this file and I list the entire file here for reference, edited to fit line width: you may find the file news.n3 easier to read if you are at your computer and open the file in a text editor so you will not be limited to what fits on a book page:

```
@prefix kb:  <http://knowledgebooks.com/ontology#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .

kb:containsCity rdfs:subPropertyOf kb:containsPlace .

kb:containsCountry rdfs:subPropertyOf kb:containsPlace .

kb:containsState rdfs:subPropertyOf kb:containsPlace .

<http://yahoo.com/20080616/usa_flooding_dc_16/>
        kb:containsCity "Burlington" , "Denver" ,
                        "St. Paul" ," Chicago" ,
                        "Quincy" , "CHICAGO" ,
                        "Iowa City" ;
        kb:containsRegion "U.S. Midwest" , "Midwest" ;
        kb:containsCountry "United States" , "Japan" ;
        kb:containsState "Minnesota" , "Illinois" ,
                         "Mississippi" , "Iowa" ;
        kb:containsOrganization "National Guard" ,
                         "U.S. Department of Agriculture" ,
                         "White House" ,
                         "Chicago Board of Trade" ,
                         "Department of Transportation" ;
        kb:containsPerson "Dena Gray-Fisher" ,
                          "Donald Miller" ,
                          "Glenn Hollander" ,
                          "Rich Feltes" ,
                          "George W. Bush" ;
        kb:containsIndustryTerm "food inflation" , "food" ,
                                "finance ministers" ,
                                "oil" .

<http://yahoo.com/78325/ts_nm/usa_politics_dc_2/>
        kb:containsCity "Washington" , "Baghdad" ,
                        "Arlington" , "Flint" ;
        kb:containsCountry "United States" ,
                           "Afghanistan" ,
                           "Iraq" ;
        kb:containsState "Illinois" , "Virginia" ,
                         "Arizona" , "Michigan" ;
        kb:containsOrganization "White House" ,
                                "Obama administration" ,
                                "Iraqi government" ;
```

```
        kb:containsPerson "David Petraeus" ,
                          "John McCain" ,
                          "Hoshiyar Zebari" ,
                          "Barack Obama" ,
                          "George W. Bush" ,
                          "Carly Fiorina" ;
        kb:containsIndustryTerm "oil prices" .

<http://yahoo.com/10944/ts_nm/worldleaders_dc_1/>
        kb:containsCity "WASHINGTON" ;
        kb:containsCountry "United States" , "Pakistan" ,
                           "Islamic Republic of Iran" ;
        kb:containsState "Maryland" ;
        kb:containsOrganization "University of Maryland" ,
                                "United Nations" ;
        kb:containsPerson "Ban Ki-moon" , "Gordon Brown" ,
                          "Hu Jintao" , "George W. Bush" ,
                          "Pervez Musharraf" ,
                          "Vladimir Putin" ,
                          "Steven Kull" ,
                          "Mahmoud Ahmadinejad" .

<http://yahoo.com/10622/global_economy_dc_4/>
        kb:containsCity "Sao Paulo" , "Kuala Lumpur" ;
        kb:containsRegion "Midwest" ;
        kb:containsCountry "United States" , "Britain" ,
                           "Saudi Arabia" , "Spain" ,
                           "Italy" , India" ,
                           ""France" , "Canada" ,
                           "Russia" , "Germany" , "China" ,
                           "Japan" , "South Korea" ;
        kb:containsOrganization "Federal Reserve Bank" ,
                                "European Union" ,
                                "European Central Bank" ,
                                "European Commission" ;
        kb:containsPerson "Lee Myung-bak" , "Rajat Nag" ,
                          "Luiz Inacio Lula da Silva" ,
                          "Jeffrey Lacker" ;
        kb:containsCompany "Development Bank Managing" ,
                           "Reuters" ,
                           "Richmond Federal Reserve Bank" ;
        kb:containsIndustryTerm "central bank" , "food" ,
                                "energy costs" ,
```

```
                "finance ministers" ,
                "crude oil prices" ,
                "oil prices" ,
                "oil shock" ,
                "food prices" ,
                "Finance ministers" ,
                "Oil prices" , "oil" .
```

In the following examples, we will use the main method in the class **JenaApi** (developed in the next section) that allows us to load multiple RDF input files and then to interactively enter SPARQL queries.

We will start with a simple SPARQL query for subjects (news article URLs) and objects (matching countries) with the value for the predicate equal to **containsCountry**. Variables in queries start with a question mark character and can have any names:

```
SELECT ?subject ?object
      WHERE {
        ?subject
        <http://knowledgebooks.com/ontology#containsCountry>
        ?object .
}
```

It is important for you to understand what is happening when we apply the last SPARQL query to our sample data. Conceptually, all the triples in the sample data are scanned, keeping the ones where the predicate part of a triple is equal to **http://knowledgebooks.com/ontology#containsCountry**. In practice RDF data stores supporting SPARQL queries index RDF data so a complete scan of the sample data is not required. This is analogous to relational databases where indices are created to avoid needing to perform complete scans of database tables.

In practice, when you are exploring a Knowledge Graph like DBPedia or WikiData (that are just very large collections of RDF triples), you might run a query and discover a useful or interesting entity URI in the triple store, then drill down to find out more about the entity. In a later chapter Knowledge Graph Navigator we attempt to automate this exploration process using the DBPedia data as a Knowledge Graph.

We will be using the same code to access the small example of RDF statements in our sample data as we will for accessing DBPedia or WikiData.

We can make this last query easier to read and reduce the chance of misspelling errors by using a namespace prefix:

```
PREFIX kb:  <http://knowledgebooks.com/ontology#>
SELECT ?subject ?object
  WHERE {
      ?subject kb:containsCountry ?object .
  }
```

Later in the chapter Knowledge Graph Navigator we will write an application that automatically generates SPARQL queries for the DBPedia public knowledge Graph. These queries will be be more complex than the simpler examples here. Reading this chapter before Knowledge Graph Navigator is recommended.

# Case Study: Using SPARQL to Find Information about Board of Directors Members of Corporations and Organizations

Before we write software to automate the process of using SPARQL queries to find information on DBPedia, let's perform a few manual queries for finding information on board of directors of corportations. To start with, we would like to find an RDF property that indicates board membership. There is a common expression for finding information on the web using search engines and also for using SPARQL queries: "follow your nose," that is, when you see something interesting, dig down with more queries on whatever interests you.

```
SELECT DISTINCT ?s
WHERE {
    ?s ?p "Board of Directors"@en .
    FILTER (?p IN (<http://www.w3.org/2000/01/rdf-schema#label>, <http://xmlns.com/f\
oaf/0.1/name>) && !regex(str(?s), "category", "i"))
}
```

We will find the property:

```
http://dbpedia.org/resource/Board_of_Directors
```

RDF

```
select ?s ?p { ?s ?p <http://dbpedia.org/resource/Board_of_Directors> } limit 6
```

```
s        p
http://en.wikipedia.org/wiki/Board_of_Directors        http://xmlns.com/foaf/0.1/primaryTop\
ic
http://dbpedia.org/resource/Lynn_D._Stewart_(businessman)        http://dbpedia.org/ontolog\
y/board
http://dbpedia.org/resource/Advance_America_Cash_Advance        http://dbpedia.org/ontology\
/keyPerson
http://dbpedia.org/resource/Railways_of_Slovak_Republic        http://dbpedia.org/ontology/\
keyPerson
http://dbpedia.org/resource/Divine_Word_University_of_Tacloban__DWU_Jubilee_Foundati\
on,_Inc.__1        http://dbpedia.org/ontology/keyPerson
http://dbpedia.org/resource/Mathys_Medical
```

The property **http://dbpedia.org/ontology/board** is what we are looking for. Let's keep "following our nose" to find examples of board members and the companies they server:

```
select ?person ?company { ?person <http://dbpedia.org/ontology/board> ?company} limi\
t 6
```

The results are:

```
person        company
http://dbpedia.org/resource/Matthew_Buckland        http://dbpedia.org/resource/Creative_Co\
mmons
http://dbpedia.org/resource/Jimmy_Wales        http://dbpedia.org/resource/Creative_Commons
http://dbpedia.org/resource/Nabeel_Rajab        http://dbpedia.org/resource/Human_Rights_Wa\
tch
http://dbpedia.org/resource/Vincent_Tewson        http://dbpedia.org/resource/International\
_Confederation_of_Free_Trade_Unions
http://dbpedia.org/resource/William_T._Young        http://dbpedia.org/resource/KFC
http://dbpedia.org/resource/Colonel_Sanders        http://dbpedia.org/resource/KFC
```

Let's see what information we can find on the founder of WikiPedi Jimmy Wales:

```
select ?p ?o { <http://dbpedia.org/resource/Jimmy_Wales> ?p ?o } limit 200
```

A few of the many results are:

```
p        o
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://www.w3.org/2002/07/owl#Thing
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://xmlns.com/foaf/0.1/Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://dbpedia.org/ontology/Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://www.wikidata.org/entity/Q2156\
27
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://www.wikidata.org/entity/Q2422\
9398
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://www.wikidata.org/entity/Q5
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://dbpedia.org/ontology/Agent
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://schema.org/Person
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://dbpedia.org/class/yago/Wikica\
tAmericanComputerScientists
http://www.w3.org/1999/02/22-rdf-syntax-ns#type        http://dbpedia.org/class/yago/Wikica\
tAmericanExpatriatesInTheUnitedKingdom
http://www.w3.org/2000/01/rdf-schema#label
"Jimmy Wales"@en
```

# Installing the Apache Jena Fuseki RDF Server

TBD

I have a github repository mark-watson/fuseki-semantic-web-dev-setup[74]mthat you shoud clone:

```
1 git clone https://github.com/mark-watson/fuseki-semantic-web-dev-setup.git
2 cd fuseki-semantic-web-dev-setup
3 ./fuseki-server --file RDF/sample_news.nt /news
```

This will run the SPARQL server Fuseki locally on your laptop and the default graph is “news” and you will see output like:

[74] https://github.com/mark-watson/fuseki-semantic-web-dev-setup

```
$ ./fuseki-server --file RDF/sample_news.nt /news
[2020-11-07 09:31:13] Server     INFO  Dataset: in-memory: load file: RDF/sample_new\
s.nt
[2020-11-07 09:31:14] Server     INFO  Running in read-only mode for /news
[2020-11-07 09:31:14] Server     INFO  Apache Jena Fuseki 3.16.0
[2020-11-07 09:31:14] Config     INFO  FUSEKI_HOME=/Users/markw/GITHUB/fuseki-semant\
ic-web-dev-setup/.
[2020-11-07 09:31:14] Config     INFO  FUSEKI_BASE=/Users/markw/GITHUB/fuseki-semant\
ic-web-dev-setup/run
[2020-11-07 09:31:14] Config     INFO  Shiro file: file:///Users/markw/GITHUB/fuseki\
-semantic-web-dev-setup/run/shiro.ini
[2020-11-07 09:31:15] Server     INFO  Dataset Type: in-memory, with files loaded
[2020-11-07 09:31:15] Server     INFO  Path = /news
[2020-11-07 09:31:15] Server     INFO  System
[2020-11-07 09:31:15] Server     INFO    Memory: 4.0 GiB
[2020-11-07 09:31:15] Server     INFO    Java:   14.0.1
[2020-11-07 09:31:15] Server     INFO    OS:     Mac OS X 10.15.7 x86_64
[2020-11-07 09:31:15] Server     INFO    PID:    3855
[2020-11-07 09:31:15] Server     INFO  Started 2020/11/07 09:31:15 MST on port 3030
```

You can access a web interface for SPARQL queries by accessing localhost:3030 or http:127.0.0.1:3030.

## Common Lisp Client Examples for the Apache Jena Fuseki RDF Server

Later in the chapter "Knowledge Graph Navigator" we will develop a simple Common Lisp SPARQL query library and use it for querying DBPedia. Here we will use it to query our local Fuseki server.

```
$ sbcl
This is SBCL 2.0.7, an implementation of ANSI Common Lisp.
More information about SBCL is available at <http://www.sbcl.org/>.

SBCL is free software, provided as is, with absolutely no warranty.
It is mostly in the public domain; some portions are provided under
BSD-style licenses.  See the CREDITS and COPYING files in the
distribution for more information.

#P"/Users/markw/quicklisp/setup.lisp"
"starting up quicklisp"
* (quicklisp:quickload "sparql")
To load "sparql":
```

```
  Load 1 ASDF system:
    sparql
; Loading "sparql"
.............
("sparql")
* (sparql::fuseki "select ?s ?p ?o { ?s ?p ?o } limit 20")
(((:s "http://kbsportal.com/trout_season/")
  (:p "http://knowledgebooks.com/ontology/#storyType")
  (:o "http://knowledgebooks.com/ontology/#recreation"))
 ((:s "http://kbsportal.com/trout_season/")
  (:p "http://knowledgebooks.com/ontology/#storyType")
  (:o "http://knowledgebooks.com/ontology/#sports"))
 ((:s "http://kbsportal.com/bear_mountain_fire/")
  (:p "http://knowledgebooks.com/ontology/#storyType")
  (:o "http://knowledgebooks.com/ontology/#disaster"))
 ((:s "http://kbsportal.com/bear_mountain_fire/")
  (:p "http://knowledgebooks.com/ontology/#summary")
  (:o "The fire on Bear Mountain was caused by lightening"))
 ((:s "http://kbsportal.com/jc_basketball/")
  (:p "http://knowledgebooks.com/ontology/#storyType")
  (:o "http://knowledgebooks.com/ontology/#sports"))
 ((:s "http://kbsportal.com/oak_creek_flooding/")
  (:p "http://knowledgebooks.com/ontology/#storyType")
  (:o "http://knowledgebooks.com/ontology/#disaster"))
 ((:s "http://kbsportal.com/oak_creek_flooding/")
  (:p "http://knowledgebooks.com/ontology/#summary")
  (:o "Oak Creek flooded last week affecting 5 businesses")))
```

Here is an example of using the same library to query the public DBPedia SPARQL endpoint (most output is not shown):

```
* (sparql:dbpedia "select ?s ?p { ?s ?p \"Bill Gates\"@en }")

("ndbpeia SPARQL:n" "select ?s ?p { ?s ?p \"Bill Gates\"@en }" "n")
(((:s "http://dbpedia.org/resource/Category:Bill_Gates")
  (:p "http://www.w3.org/2000/01/rdf-schema#label"))
 ((:s "http://www.wikidata.org/entity/Q5284")
  (:p "http://www.w3.org/2000/01/rdf-schema#label"))
 ((:s "http://dbpedia.org/resource/Bill_Gates")
  (:p "http://xmlns.com/foaf/0.1/name"))
 )
```

The SPARQL library in the github repository for this book also supports the commercial products

AllegroGraph and Stardog RDF servers.

# Automatically Generating Data for Knowledge Graphs

We develop a complete application. The Knowledge Graph Creator (KGcreator) is a tool for automating the generation of data for Knowledge Graphs from raw text data. We will see how to create a single standalone executable file using SBCL Common Lisp. The application can also be run during development from a repl. This application also implements a web application interface. In addition to the KGcreator application we will close the chapter with a utiity library that processes a file of RDF in N-Triple format and generates an extention file with triples pulled from DBedia defining URIs found in the input data file.

Data created by KGcreator generates data in two formats:

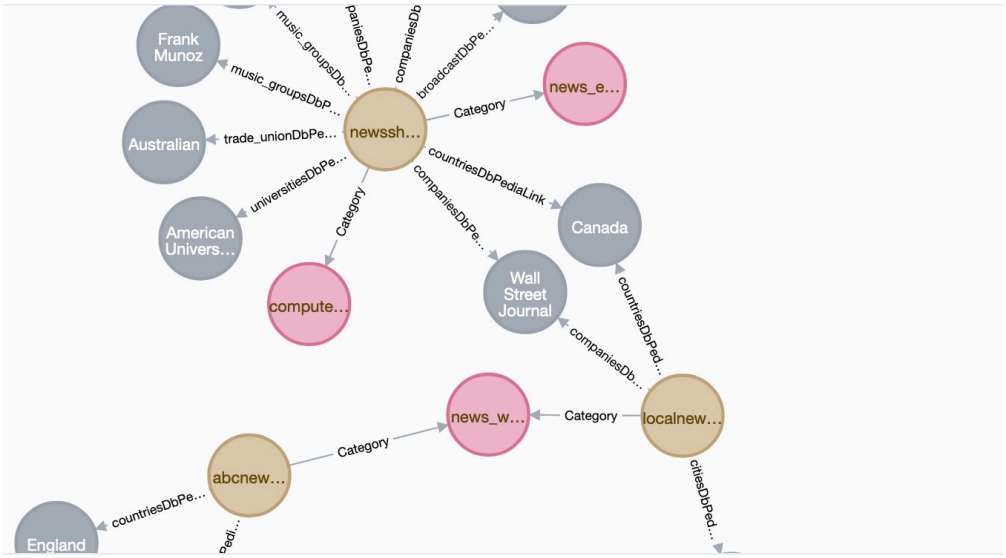
- Neo4j graph database format (text format)
- RDF triples suitable for loading into any linked data/semantic web data store.

This example application works by identifying entities in text. Example entity types are people, companies, country names, city names, broadcast network names, political party names, and university names. We saw earlier code for detecting entities in the chapter on natural language processing (NLP) and we will reuse this code. We will discuss later three strategies for reusing code from different projects.

When I originally wrote KGCreator I intended to develop a commercial product. I wrote two research prototypes, one in Common Lisp (the example in this chapter) and one in Haskell (which I also use as an example in my book Haskell Tutorial and Cookbook[75]. I decided to open source both versions of KGCreator and if you work with Knowledge Graphs I hope you find KGCreator useful in your work.

The following figure shows part of a Neo4j Knowledge Graph created with the example code. This graph has shortened labels in displayed nodes but Neo4j offers a web browser-based console that lets you interactively explore Knowledge Graphs. We don't cover setting up Neo4j here so please use the Neo4j documentation[76]. As an introduction to RDF data, the semantic web, and linked data you can get free copies of my two books Practical Semantic Web and Linked Data Applications, Common Lisp Edition[77] and Practical Semantic Web and Linked Data Applications, Java, Scala, Clojure, and JRuby Edition[78].

[75] https://leanpub.com/haskell-cookbook/
[76] https://neo4j.com/docs/operations-manual/current/introduction/
[77] http://markwatson.com/opencontentdata/book_lisp.pdf
[78] http://markwatson.com/opencontentdata/book_java.pdf




**Part of a Knowledge Graph shown in Neo4j web application console**

Here is a detail view:


**Detail of Neo4j console**

# Implementation Notes

As seen in the file **src /kgcreator/package.lisp** this application uses several other packages:

```
1  (defpackage #:kgcreator
2    (:use #:cl
3          #:entities_dbpedia #:categorize_summarize #:myutils
4          #:cl-who #:hunchentoot #:parenscript)
5    (:export kgcreator))
```

The implementation of the packages shown on line 3 were in a previous chapter. The package **myutils** are mostly miscellaneous string utilities that we won't look at here; I leave it to you to read the source code.

As seen in the configuration file **src/kgcreator/kgcreator.asd** we split the implementation of the application into four source files:

```
1  ;;;; kgcreator.asd
2
3  (asdf:defsystem #:kgcreator
4    :description "Describe plotlib here"
5    :author "Mark Watson <mark.watson@gmail.com>"
6    :license "AGPL version 3"
7    :depends-on (#:entities_dbpedia #:categorize_summarize
8                 #:myutils #:unix-opts #:cl-who
9                 #:hunchentoot #:parenscript)
10   :components
11     ((:file "package")
12                  (:file "kgcreator")
13                  (:file "neo4j")
14                  (:file "rdf")
15                  (:file "web"))
16                   )
```

The application is separated into four source files:

- kgcreator.lisp: top level APIs and functionality. Uses the code in neo4j.lisp and rdf.lisp. Later we will generate a standalone application that uses these top level APIs
- neo4j.lisp: generates Cyper text files that can be imported into Neo4j
- – rdf.lisp: generates RDF text data that can be loaded or imported into RDF data stores
- web.lisp: a simple web application for running KGCreator

# Generating RDF Data

I leave it to you find a tutorial on RDF data on the web, or you can get a PDF for my book "Practical Semantic Web and Linked Data Applications, Common Lisp Edition"[79] and read the tutorial sections on RDF.

[79] http://markwatson.com/opencontentdata/book_lisp.pdf

RDF data is comprised of triples, where the value for each triple are a subject, a predicate, and an object. Subjects are URIs, predicates are usually URIs, and objects are either literal values or URIs. Here are two triples written by this example application:

```
<http://dbpedia.org/resource/The_Wall_Street_Journal>
  <http://knowledgebooks.com/schema/aboutCompanyName>
  "Wall Street Journal" .
<https://newsshop.com/june/z902.html>
  <http://knowledgebooks.com/schema/containsCountryDbPediaLink>
  <http://dbpedia.org/resource/Canada> .
```

The following listing of the file **src/kgcreator/rdf.lisp** generates RDF data:

```
(in-package #:kgcreator)

(let ((*rdf-nodes-hash*))

  (defun rdf-from-files (output-file-path text-and-meta-pairs)
    (setf *rdf-nodes-hash* (make-hash-table :test #'equal :size 200))
    (print (list "==> rdf-from-files" output-file-path text-and-meta-pairs ))
    (with-open-file
        (str output-file-path
             :direction :output
             :if-exists :supersede
             :if-does-not-exist :create)

      (defun rdf-from-files-handle-single-file (text-input-file meta-input-file)
        (let* ((text (file-to-string text-input-file))
               (words (myutils:words-from-string text))
               (meta (file-to-string meta-input-file)))

          (defun generate-original-doc-node-rdf ()
            (let ((node-name (node-name-from-uri meta)))
              (if (null (gethash node-name *rdf-nodes-hash*))
                  (let* ((cats (categorize words))
                         (sum (summarize words cats)))
                    (print (list "$$$$$$  cats:" cats))
                    (setf (gethash node-name *rdf-nodes-hash*) t)
                    (format str (concatenate 'string "<" meta
                                             "> <http:knowledgebooks.com/schema/summary> \""
                                             sum "\" . ~%"))
                    (dolist (cat cats)
                      (let ((hash-check (concatenate 'string node-name (car cat))))
```

```
                        (if (null (gethash hash-check *rdf-nodes-hash*))
                            (let ()
                              (setf (gethash hash-check *rdf-nodes-hash*) t)
                              (format str
                                      (concatenate 'string "<" meta
                                                   "> <http://knowledgebooks.com/schema/"
                                                   "topicCategory> "
                                                   "<http://knowledgebooks.com/schema/"
                                                   (car cat) "> . ~%")))))))))

          (defun generate-dbpedia-contains-rdf (key value)
            (generate-original-doc-node-rdf)
            (let ((relation-name (concatenate 'string key "DbPediaLink")))
              (dolist (entity-pair value)
                (let* ((node-name (node-name-from-uri meta))
                       (object-node-name (node-name-from-uri (cadr entity-pair)))
                       (hash-check (concatenate 'string node-name object-node-name)))
                  (if (null (gethash hash-check *rdf-nodes-hash*))
                      (let ()
                        (setf (gethash hash-check *rdf-nodes-hash*) t)
                        (format str (concatenate 'string "<" meta
                                                 "> <http://knowledgebooks.com/schema/contains/"
                                                 key "> " (cadr entity-pair) " .~%")))))))))


      ;; start code for rdf-from-files (output-file-path text-and-meta-pairs)
      (dolist (pair text-and-meta-pairs)
        (rdf-from-files-handle-single-file (car pair) (cadr pair))
        (let ((h (entities_dbpedia:find-entities-in-text (file-to-string (car pair))\
)))
          (entities_dbpedia:entity-iterator #'generate-dbpedia-contains-rdf h))))))


(defvar test_files '((#P"~/GITHUB/common-lisp/kgcreator/test_data/test3.txt"
                      #P"~/GITHUB/common-lisp/kgcreator/test_data/test3.meta")))
(defvar test_filesZZZ '((#P"~/GITHUB/common-lisp/kgcreator/test_data/test3.txt"
                         #P"~/GITHUB/common-lisp/kgcreator/test_data/test3.meta")
                        (#P"~/GITHUB/common-lisp/kgcreator/test_data/test2.txt"
                         #P"~/GITHUB/common-lisp/kgcreator/test_data/test2.meta")
                        (#P"~/GITHUB/common-lisp/kgcreator/test_data/test1.txt"
                         #P"~/GITHUB/common-lisp/kgcreator/test_data/test1.meta")))

(defun test3a ()
```

```
74    (rdf-from-files "out.rdf" test_files))
```

You can load all of KGCreator but just execute the test function at the end of this file using:

```
(ql:quickload "kgcreator")
(in-package #:kgcreator)
(kgcreator:test3a)
```

This code works on a list of paired files for text data and the meta data for each text file. As an example, if there is an input text file test123.txt then there would be a matching meta file test123.meta that contains the source of the data in the file test123.txt. This data source will be a URI on the web or a local file URI. The top level function **rdf-from-files** takes an output file path for writing the generated RDF data and a list of pairs of text and meta file paths.

A global variable ***rdf-nodes-hash*** will be used to remember the nodes in the RDF graph as it is generated. Please note that the function **rdf-from-files** is not re-entrant: it uses the global ***rdf-nodes-hash*** so if you are writing multi-threaded applications it will not work to execute the function **rdf-from-files** simultaneously in multiple threads of execution.

The function **rdf-from-files** (and the nested functions) are straightforward. I left a few debug printout statements in the code and when you run the test code that I left in the bottom of the file, hopefully it will be clear what rdf.lisp is doing.

# Generating Data for the Neo4j Graph Database

Now we will generate Neo4J Cypher data. In order to keep the implementation simple, both the RDF and Cypher generation code starts with raw text and performs the NLP analysis to find entities. This example could be refactored to perform the NLP analysis just one time but in practice you will likely be working with either RDF or NEO4J and so you will probably extract just the code you need from this example (i.e., either the RDF or Cypher generation code).

Before we look at the code, let's start with a few lines of generated Neo4J Cypher import data:

```
CREATE (newsshop_com_june_z902_html_news)-[:ContainsCompanyDbPediaLink]->(Wall_Stree\
t_Journal)
CREATE (Canada:Entity {name:"Canada", uri:"<http://dbpedia.org/resource/Canada>"})
CREATE (newsshop_com_june_z902_html_news)-[:ContainsCountryDbPediaLink]->(Canada)
CREATE (summary_of_abcnews_go_com_US_violent_long_lasting_tornadoes_threaten_oklahom\
a_texas_storyid63146361:Summary {name:"summary_of_abcnews_go_com_US_violent_long_las\
ting_tornadoes_threaten_oklahoma_texas_storyid63146361", uri:"<https://abcnews.go.co\
m/US/violent-long-lasting-tornadoes-threaten-oklahoma-texas/story?id=63146361>", sum\
mary:"Part of the system that delivered severe weather to the central U.S. over the \
weekend is moving into the Northeast today, producing strong to severe storms -- dam\
```

```
aging winds, hail or isolated tornadoes can't be ruled out. Severe weather is foreca\
st to continue on Tuesday, with the western storm moving east into the Midwest and p\
arts of the mid-Mississippi Valley."})
```

The following listing of file **src/kgcreator/neo4j.lisp** is similar to the code that generated RDF in the last section:

```
(in-package #:kgcreator)

(let ((*entity-nodes-hash*))

  (defun cypher-from-files (output-file-path text-and-meta-pairs)
    (setf *entity-nodes-hash* (make-hash-table :test #'equal :size 200))
    ;;(print (list "==> cypher-from-files"output-file-path text-and-meta-pairs ))
    (with-open-file
        (str output-file-path
             :direction :output
             :if-exists :supersede
             :if-does-not-exist :create)

      (defun generateNeo4jCategoryNodes ()
        (let* ((names categorize_summarize::categoryNames))
          (dolist (name names)
            (format str
                    (myutils:replace-all
                     (concatenate
                      'string "CREATE (" name ":CategoryType {name:\"" name "\"})~%")
                     "/" "_"))))
        (format str "~%"))


      (defun cypher-from-files-handle-single-file (text-input-file meta-input-file)
        (let* ((text (file-to-string text-input-file))
               (words (myutils:words-from-string text))
               (meta (file-to-string meta-input-file)))

          (defun generate-original-doc-node ()
            (let ((node-name (node-name-from-uri meta)))
              (if (null (gethash node-name *entity-nodes-hash*))
                  (let* ((cats (categorize words))
                         (sum (summarize words cats)))
                    (setf (gethash node-name *entity-nodes-hash*) t)
                    (format str (concatenate 'string "CREATE (" node-name ":News {name:\""
```

```
                                          node-name "\", uri: \"" meta
              "\", summary: \"" sum "\"})~%"))
                   (dolist (cat cats)
                     (let ((hash-check (concatenate 'string node-name (car cat))))
                       (if (null (gethash hash-check *entity-nodes-hash*))
                           (let ()
                             (setf (gethash hash-check *entity-nodes-hash*) t)
                             (format str (concatenate 'string "CREATE (" node-name
                                    ")-[:Category]->("
                                                (car cat) ")~%")))))))))

        (defun generate-dbpedia-nodes (key entity-pairs)
          (dolist (entity-pair entity-pairs)
            (if (null (gethash (node-name-from-uri (cadr entity-pair))
                               *entity-nodes-hash*))
                (let ()
                  (setf (gethash (node-name-from-uri (cadr entity-pair)) *entity-nodes-hash*) t)
                  (format str
                          (concatenate 'string "CREATE (" (node-name-from-uri (cadr entity-pair))
                                       key " {name: \"" (car entity-pair)
                                       "\", uri: \"" (cadr entity-pair) "\"})~%"))))))

        (defun generate-dbpedia-contains-cypher (key value)
          (generate-original-doc-node)
          (generate-dbpedia-nodes key value)
          (let ((relation-name (concatenate 'string key "DbPediaLink")))
            (dolist (entity-pair value)
              (let* ((node-name (node-name-from-uri meta))
                     (object-node-name (node-name-from-uri (cadr entity-pair)))
                     (hash-check (concatenate 'string node-name object-node-name)))
                (if (null (gethash hash-check *entity-nodes-hash*))
                    (let ()
                      (setf (gethash hash-check *entity-nodes-hash*) t)
                      (format str (concatenate 'string
                                               "CREATE (" node-name ")-[:"
                                               relation-name "]->(" object-node-name ")~%")))))))))


    ;; start code for cypher-from-files (output-file-path text-and-meta-pairs)
    (generateNeo4jCategoryNodes) ;; just once, not for every input file
    (dolist (pair text-and-meta-pairs)
      (cypher-from-files-handle-single-file (car pair) (cadr pair))
      (let ((h (entities_dbpedia:find-entities-in-text (file-to-string (car pair)))))
```

```
          (entities_dbpedia:entity-iterator #'generate-dbpedia-contains-cypher h))))))


(defvar test_files '((#P"~/GITHUB/common-lisp/kgcreator/test_data/test3.txt"
                      #P"~/GITHUB/common-lisp/kgcreator/test_data/test3.meta")
                     (#P"~/GITHUB/common-lisp/kgcreator/test_data/test2.txt"
                      #P"~/GITHUB/common-lisp/kgcreator/test_data/test2.meta")
                     (#P"~/GITHUB/common-lisp/kgcreator/test_data/test1.txt"
                      #P"~/GITHUB/common-lisp/kgcreator/test_data/test1.meta")))

(defun test2a ()
  (cypher-from-files "out.cypher" test_files))
```

You can load all of KGCreator but just execute the test function at the end of this file using:

```
(ql:quickload "kgcreator")
(in-package #:kgcreator)
(kgcreator:test2a)
```

# Implementing the Top Level Application APIs

The code in the file **src/kgcreator/kgcreator.lisp** uses both rdf.lisp and neo4j.lisp that we saw in the last two sections. The function **get-files-and-meta** looks at the contents of an input directory to generate a list of pairs, each pair containing the path to a text file and the meta file for the corresponding text file.

We are using the **opts** package to parse command line arguments. This will be used when we build a single file standalone executable file for the entire KGCreator application, including the web application that we will see in a later section.

```
;; KGCreator main program

(in-package #:kgcreator)

(ensure-directories-exist "temp/")

(defun get-files-and-meta (fpath)
  (let ((data (directory (concatenate 'string fpath "/" "*.txt")))
              (meta (directory (concatenate 'string fpath "/" "*.meta"))))
    (if (not (equal (length data) (length meta)))
        (let ()
```

```
          (princ "Error: must be matching *.meta files for each *.txt file")
          (terpri)
          '())
        (let ((ret '()))
          (dotimes (i (length data))
            (setq ret (cons (list (nth i data) (nth i meta)) ret)))
          ret))))

(opts:define-opts
    (:name :help
           :description
     "KGcreator command line example: ./KGcreator -i test_data -r out.rdf -c out.cyp\
er"
           :short #\h
           :long "help")
    (:name :rdf
           :description "RDF output file name"
           :short #\r
           :long "rdf"
           :arg-parser #'identity ;; <- takes an argument
           :arg-parser #'identity) ;; <- takes an argument
  (:name :cypher
         :description "Cypher output file name"
         :short #\c
         :long "cypher"
         :arg-parser #'identity) ;; <- takes an argument
  (:name :inputdir
         :description "Cypher output file name"
         :short #\i
         :long "inputdir"
         :arg-parser #'identity)) ;; <- takes an argument


(defun kgcreator () ;; don't need: &aux args sb-ext:*posix-argv*)
  (handler-case
      (let* ((opts (opts:get-opts))
             (input-path
              (if (find :inputdir opts)
                  (nth (1+ (position :inputdir opts)) opts)))
             (rdf-output-path
              (if (find :rdf opts)
                  (nth (1+ (position :rdf opts)) opts)))
             (cypher-output-path
```

```
55                (if (find :cypher opts)
56                    (nth (1+ (position :cypher opts)) opts))))
57          (format t "input-path: ~a  rdf-output-path: ~a cypher-output-path:~a~%"
58                  input-path rdf-output-path cypher-output-path)
59          (if (not input-path)
60              (format t "You must specify an input path.~%")
61              (locally
62                  (declare #+sbcl(sb-ext:muffle-conditions sb-kernel:redefinition-warning))
63                (handler-bind
64                    (#+sbcl(sb-kernel:redefinition-warning #'muffle-warning))
65                  ;; stuff that emits redefinition-warning's
66                  (let ()
67                    (if rdf-output-path
68                        (rdf-from-files rdf-output-path (get-files-and-meta input-path)))
69                    (if cypher-output-path
70                        (cypher-from-files cypher-output-path (get-files-and-meta input-path))))))))
71      (t (c)
72        (format t "We caught a runtime error: ~a~%" c)
73        (values 0 c)))
74    (format t "~%Shutting down KGcreator - done processing~%~%"))
75
76  (defun test1 ()
77      (get-files-and-meta
78       "~/GITHUB/common-lisp/kgcreator/test_data"))
79
80  (defun print-hash-entry (key value)
81    (format t "The value associated with the key ~S is ~S~%" key value))
82
83  (defun test2 ()
84    (let ((h (entities_dbpedia:find-entities-in-text "Bill Clinton and George Bush wen\
85  t to Mexico and England and watched Univision. They enjoyed Dakbayan sa Dabaw and sh\
86  oped at Best Buy and listened to Al Stewart. They agree on RepÃºblica de Nicaragua a\
87  nd support Sweden Democrats and Leicestershire Miners Association and both sent thei\
88  r kids to Darul Uloom Deoband.")))
89      (entities_dbpedia:entity-iterator #'print-hash-entry h)))
90
91  (defun test7 ()
92    (rdf-from-files "out.rdf" (get-files-and-meta "test_data")))
```

You can load all of KGCreator but just execute the three test functions at the end of this file using:

```
(ql:quickload "kgcreator")
(in-package #:kgcreator)
(kgcreator:test1)
(kgcreator:test2)
(kgcreator:test7)
```

# Implementing The Web Interface

When we build a standalone single file application for KGCreator, we include a simple web application interface that allows users to enter input text and see generated RDF and Neo4j Cypher data.

The file **src/kgcreator/web.lisp** uses the libraries **cl-who hunchentoot parenscript** that we used earlier. The function write-files-run-code** (lines 8-43) takes raw text, and writes generated RDF and Neo4j Cypher data to local temporary files that are then read and formatted to HTML for display. The code in rdf.lisp and neo4j.lisp is file oriented, and I wrote web.lisp as an afterthought so it was easier writing temporary files than refactoring rdf.lisp and neo4j.lisp to write to strings.

```
1  (in-package #:kgcreator)
2
3  (ql:quickload '(cl-who hunchentoot parenscript))
4
5
6  (setf (html-mode) :html5)
7
8  (defun write-files-run-code (a-uri raw-text)
9    (if (< (length raw-text) 10)
10       (list "not enough text" "not enough text")
11     ;; generate random file number
12     (let* ((filenum (+ 1000 (random 5000)))
13            (meta-name (concatenate 'string "temp/" (write-to-string filenum) ".meta"))
14            (text-name (concatenate 'string "temp/" (write-to-string filenum) ".txt"))
15            (rdf-name (concatenate 'string "temp/" (write-to-string filenum) ".rdf"))
16            (cypher-name (concatenate 'string "temp/" (write-to-string filenum) ".cypher"))
17            ret)
18       ;; write meta file
19       (with-open-file (str meta-name
20                            :direction :output
21                            :if-exists :supersede
22                            :if-does-not-exist :create)
23                       (format str a-uri))
24       ;; write text file
```

```
      (with-open-file (str text-name
                           :direction :output
                           :if-exists :supersede
                           :if-does-not-exist :create)
        (format str raw-text))
      ;; generate rdf and cypher files
      (rdf-from-files rdf-name (list (list text-name meta-name)))
      (cypher-from-files cypher-name (list (list text-name meta-name)))
      ;; read files and return results
      (setf ret
            (list
             (replace-all
              (replace-all
               (uiop:read-file-string rdf-name)
               ">" "&gt;")
              "<" "&lt;")
             (uiop:read-file-string cypher-name)))
      (print (list "ret:" ret))
      ret)))

(defvar *h* (make-instance 'easy-acceptor :port 3000))

;; define a handler with the arbitrary name my-greetings:

(define-easy-handler (my-greetings :uri "/") (text)
  (setf (hunchentoot:content-type*) "text/html")
  (let ((rdf-and-cypher (write-files-run-code "http://test.com/1" text)))
    (print (list "*** rdf-and-cypher:" rdf-and-cypher))
    (with-html-output-to-string
     (*standard-output* nil :prologue t)
     (:html
      (:head (:title "KGCreator Demo")
             (:link :rel "stylesheet" :href "styles.css" :type "text/css"))
      (:body
       :style "margin: 90px"
       (:h1 "Enter plain text for the demo to create RDF and Cypher")
       (:p "For more information on the KGCreator product please visit the web site:"
           (:a :href "https://markwatson.com/products/" "Mark Watson's commercial products"\
))
       (:p "The KGCreator product is a command line tool that processes all text "
           "web applications and files in a source directory and produces both RDF data "
           "triples for semantic Cypher input data files for the Neo4j graph database. "
           "For the purposes of this demo the URI for your input text is hardwired to "
```

```
            "&lt;http://test.com/1&gt; but the KGCreator product offers flexibility "
                  "for assigning URIs to data sources and further, "
            "creates links for relationships between input sources.")
        (:p :style "text-align: left"
            "To try the demo paste plain text into the following form that contains "
                  "information on companies, news, politics, famous people, broadcasting "
                  "networks, political parties, countries and other locations, etc. ")
        (:p "Do not include and special characters or character sets:")
        (:form
         :method :post
         (:textarea
          :rows "20"
          :cols "90"
          :name "text"
          :value text)
         (:br)
         (:input :type :submit :value "Submit text to process"))
        (:h3 "RDF:")
        (:pre (str (car rdf-and-cypher)))
        (:h3 "Cypher:")
        (:pre (str (cadr rdf-and-cypher))))))))

(defun kgcweb ()
  (hunchentoot:start *h*))
```

You can load all of KGCreator and start the web application using:

```
(ql:quickload "kgcreator")
(in-package #:kgcreator)
(kgcweb)
```

You can access the web app at http://localhost:3000[80].

## Creating a Standalone Application Using SBCL

When I originally wrote KGCreator I intended to develop a commercial product so it was important to be able to create standalone single file executables. This is simple to do using SBCL:

[80] http://localhost:3000

```
$ sbcl
(ql:quickload "kgcreator")
(in-package #:kgcreator)
(sb-ext:save-lisp-and-die "KGcreator"
  :toplevel #'kgcreator :executable t)
```

As an example, you could run the application on the command line using:

```
./KGcreator -i test_data -r out.rdf -c out.cyper
```

# Augmenting RDF Triples in a Knowledge Graph Using DBPedia

You can augment RDF-based Knowledge Graphs that you build with the KGcreator application by using the library in the directory **kg-add-dbpedia-triples**.

As seen in the **kg-add-dbpedia-triples.asd** and **package.lisp** configuration files, we use two other libraries developed in this book:

```
;;;; kg-add-dbpedia-triples.asd

(asdf:defsystem #:kg-add-dbpedia-triples
  :description "Add DBPedia triples from an input N-Triples RDF file"
  :author "markw@markwatson.com"
  :license "Apache 2"
  :depends-on (#:myutils #:sparql)
  :components ((:file "package")
               (:file "add-dbpedia-triples")))


;;;; package.lisp

(defpackage #:kg-add-dbpedia-triples
  (:use #:cl #:myutils #:sparql)
  (:export #:add-triples))
```

The library is implemented in the file **kg-add-dbpedia-triples.lisp**:

```
(in-package #:kg-add-dbpedia-triples)

(defun augmented-triples (a-uri ostream)
  (let ((results
         (sparql:dbpedia
          (format nil "construct { ~A ?p ?o } where { ~A ?p ?o } limit 5" a-uri a-ur\
i))))
    (dolist (x results)
      (dolist (sop x)
        (let ((val (second sop)))
          (if (and
               (stringp val)
               (> (length val) 9)
               (or
                (equal (subseq val 0 7) "http://")
                (equal (subseq val 0 8) "https://")))
              (format ostream "<~A> " val)
              (format ostream "~A " val))))
      (format ostream " .~%"))))

(defun add-triples (in-file-name out-file-name)
  (let* ((nt-data (myutils:file-to-string in-file-name))
         (tokens (myutils:tokenize-string-keep-uri nt-data))
         (uris
          (remove-duplicates
           (mapcan #'(lambda (s) (if
                                  (and
                                   (stringp s)
                                   (> (length s) 19)
                                   (equal (subseq s 0 19) "<http://dbpedia.org"))
                                  (list s)))
                   tokens)
           :test #'equal)))
    (with-open-file (str out-file-name
                         :direction :output
                         :if-exists :supersede
                         :if-does-not-exist :create)
    (dolist (uri uris)
      (augmented-triples uri str)))))
```

TBD

```
1
```

## KGCreator Wrap Up

When developing applications or systems using Knowledge Graphs it is useful to be able to quickly generate test data which is the primary purpose of KGCreator. A secondary use is to generate Knowledge Graphs for production use using text data sources. In this second use case you will want to manually inspect the generated data to verify its correctness or usefulness for your application.

# Knowledge Graph Sampler for Creating Small Custom Knowledge Graphs

I find it convenient to be able to "sample" small parts of larger knowledge graphs. The example program in this chapter accepts a list of DBPedia entity URIs, attempts top find links between these entities, and writes these nodes and discovered edges to a RDF triples file.

The code is in the directory **src/kgsampler**. As seen in the configuration files **kg-add-dbpedia-triples.asd** and **package.lisp**, we will use the **sparql** library we developed earlier as well as the libraries **uiop** and **drakma**:

```
;;;; kgsampler.asd

(asdf:defsystem #:kgsampler
  :description "sample knowledge graphs"
  :author "Mark Watson markw@markwatson.com"
  :license "Apache 2"
  :depends-on (#:uiop #:drakma #:sparql)
  :components ((:file "package")
               (:file "kgsampler")))


;;;; package.lisp

(defpackage #:kgsampler
  (:use #:cl #:uiop #:sparql)
  (:export #:sample))
```

The program starts with a list of entities and tries to find links on DBPedia between the entities. A small sample graph of the input entities and any discovered links is written to a file. The function **dbpedia-as-nt** spawns a process to use the *curl* utility to make a HTTP request to DBPedia. The function **construct-from-dbpedia** takes a list of entities and writes SPARQL CONSTRUCT statements with the entity as the subject and the object filtered to a string value in the English language to an output stream. The function **find-relations** runs at **O(N^2)** where **N** is the number of input entities so you should avoid using this program with a large number of input entities.

I offer this code without much explanation since much of it is similar to the techniques you saw in the previous chapter Knowledge Graph Navigator.

```
;; kgsampler main program

(in-package #:kgsampler)

(defun dbpedia-as-nt (query)
  (print query)
  (uiop:run-program
   (list
    "curl"
    (concatenate 'string
                 "https://dbpedia.org/sparql?format=text/ntriples&query="
                 ;; formats that work: csv, text/ntriples, text/ttl
                 (drakma:url-encode query :utf-8)))
   :output :string))

(defun construct-from-dbpedia (entity-uri-list &key (output-stream t))
  (dolist (entity-uri entity-uri-list)
    (format output-stream "~%~%# ENTITY NAME: ~A~%~%" entity-uri)
    (format
     output-stream
     (dbpedia-as-nt
      (format nil
              "CONSTRUCT { ~A ?p ?o } where { ~A ?p ?o  . FILTER(lang(?o) = 'en') }"\

              entity-uri entity-uri)))))

(defun ensure-angle-brackets (s)
  "make sure URIs have angle brackets"
  (if (equal #\< (char s 0))
      s
      (concatenate 'string "<" s ">")))

(defun find-relations (entity-uri-list &key (output-stream t))
  (dolist (entity-uri1 entity-uri-list)
    (dolist (entity-uri2 entity-uri-list)
      (if (not (equal entity-uri1 entity-uri2))
          (let ((possible-relations
                 (mapcar #'cadar
                         (sparql::dbpedia
                          (format nil
                                  "select ?p where { ~A ?p ~A . filter(!regex(str(?p\
), \"page\", \"i\"))} limit 50"
                                  entity-uri1 entity-uri2)))))
```

```
            (print "** possible-relations:") (print possible-relations)
            (dolist (pr possible-relations)
              (format output-stream "~A ~A ~a .~%"
                      entity-uri1
                      (ensure-angle-brackets pr)
                      entity-uri2)))))))

(defun sample (entity-uri-list output-filepath)
  (with-open-file (ostream  (pathname output-filepath) :direction :output :if-exists\
 :supersede)
    (construct-from-dbpedia entity-uri-list :output-stream ostream)
    (find-relations entity-uri-list :output-stream ostream)))
```

Let's start by running the two helper functions interactively so you can see their output (output edited for brevity). The top level function **kgsampler:sample** for this example takes a list of entity URIs and an output file name, and uses the functions **construct-from-dbpedia entity-uri-list** and **find-relations** to write triples for the entities and then for the relationships discovered between entities. The following listing also calls the helper function **kgsampler::find-relations** to show you what its output looks like.

```
$ sbcl
* (ql:quickload "kgsampler")
*   (kgsampler::construct-from-dbpedia '("<http://dbpedia.org/resource/Bill_Gates>" \
"<http://dbpedia.org/resource/Steve_Jobs>") :output-stream nil)

"CONSTRUCT { <http://dbpedia.org/resource/Bill_Gates> ?p ?o } where { <http://dbpedi\
a.org/resource/Bill_Gates> ?p ?o  . FILTER (lang(?o) = 'en') }"
"CONSTRUCT { <http://dbpedia.org/resource/Bill_Gates> <http://purl.org/dc/terms/subj\
ect> ?o } where { <http://dbpedia.org/resource/Bill_Gates> <http://purl.org/dc/terms\
/subject> ?o  }"

 ...

* (kgsampler::find-relations '("<http://dbpedia.org/resource/Bill_Gates>" "<http://d\
bpedia.org/resource/Microsoft>") :output-stream nil)

("dbpedia SPARQL:"
 "select ?p where { <http://dbpedia.org/resource/Bill_Gates> ?p <http://dbpedia.org/\
resource/Microsoft> . filter(!regex(str(?p), \"page\", \"i\"))} limit 50"
 "n")
"** possible-relations:"
("http://dbpedia.org/ontology/knownFor")
"http://dbpedia.org/ontology/knownFor"
```

```
("dbpedia SPARQL:"
 "select ?p where { <http://dbpedia.org/resource/Microsoft> ?p <http://dbpedia.org/r\
esource/Bill_Gates> . filter(!regex(str(?p), \"page\", \"i\"))} limit 50"
 "n")
"** possible-relations:"
("http://dbpedia.org/property/founders" "http://dbpedia.org/ontology/foundedBy")
"http://dbpedia.org/property/founders"
"http://dbpedia.org/ontology/foundedBy"
nil
```

We now use the main function to generate an output RDF triple file:

```
$ sbcl
* (ql:quickload "kgsampler")
* (kgsampler:sample '("<http://dbpedia.org/resource/Bill_Gates>" "<http://dbpedia.or\
g/resource/Steve_Jobs>" "<http://dbpedia.org/resource/Microsoft>")  "test.nt")
"CONSTRUCT { <http://dbpedia.org/resource/Bill_Gates> ?p ?o } where { <http://dbpedi\
a.org/resource/Bill_Gates> ?p ?o  . FILTER (lang(?o) = 'en') }"
("ndbpedia SPARQL:n"
 "select ?p where { <http://dbpedia.org/resource/Bill_Gates> ?p <http://dbpedia.org/\
resource/Microsoft> . filter(!regex(str(?p), \"page\", \"i\"))} limit 50"
 "n")
"** possible-relations:"
("http://dbpedia.org/ontology/board")
("dbpedia SPARQL:"
 "select ?p where { <http://dbpedia.org/resource/Steve_Jobs> ?p <http://dbpedia.org/\
resource/Bill_Gates> . filter(!regex(str(?p), \"page\", \"i\"))} limit 50"
 "n")
```

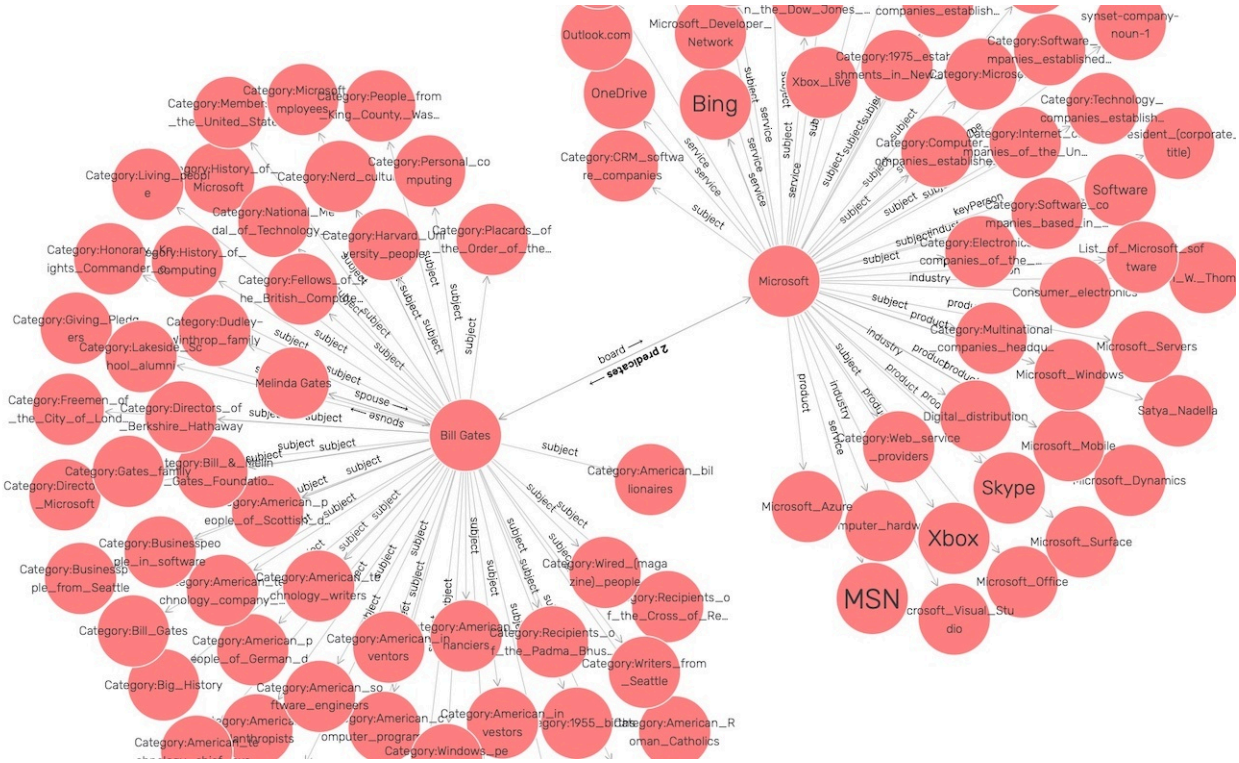
Output RDF N-Triple data is written to the file **sample-KG.nt**. A very small part of this file is listed here:

```
# ENTITY NAME: <http://dbpedia.org/resource/Bill_Gates>

<http://dbpedia.org/resource/Bill_Gates>        <http://dbpedia.org/ontology/abstrac\
t>  "William Henry \"Bill\" Gates III (born October 28, 1955) is an American busines\
s magnate,...."@en .
<http://dbpedia.org/resource/Bill_Gates>
        <http://xmlns.com/foaf/0.1/name>
        "Bill Gates"@en .
<http://dbpedia.org/resource/Bill_Gates>
        <http://xmlns.com/foaf/0.1/surname>
```

```
        "Gates"@en .
<http://dbpedia.org/resource/Bill_Gates>
        <http://dbpedia.org/ontology/title>
        "Co-Chairmanof theBill & Melinda Gates Foundation"@en .
```

The same data in Turtle RDF format can be seen in the file **sample-KG.ttl** that was produced by importing the triples file into the free edition of GraphDB exporting it to the Turtle file **sample-KG.ttl** that I find easier to read. GraphDB has visualization tools which I use here to generate an interactive graph display of this data:




**GraphDB Visual graph of generated RDF triples**

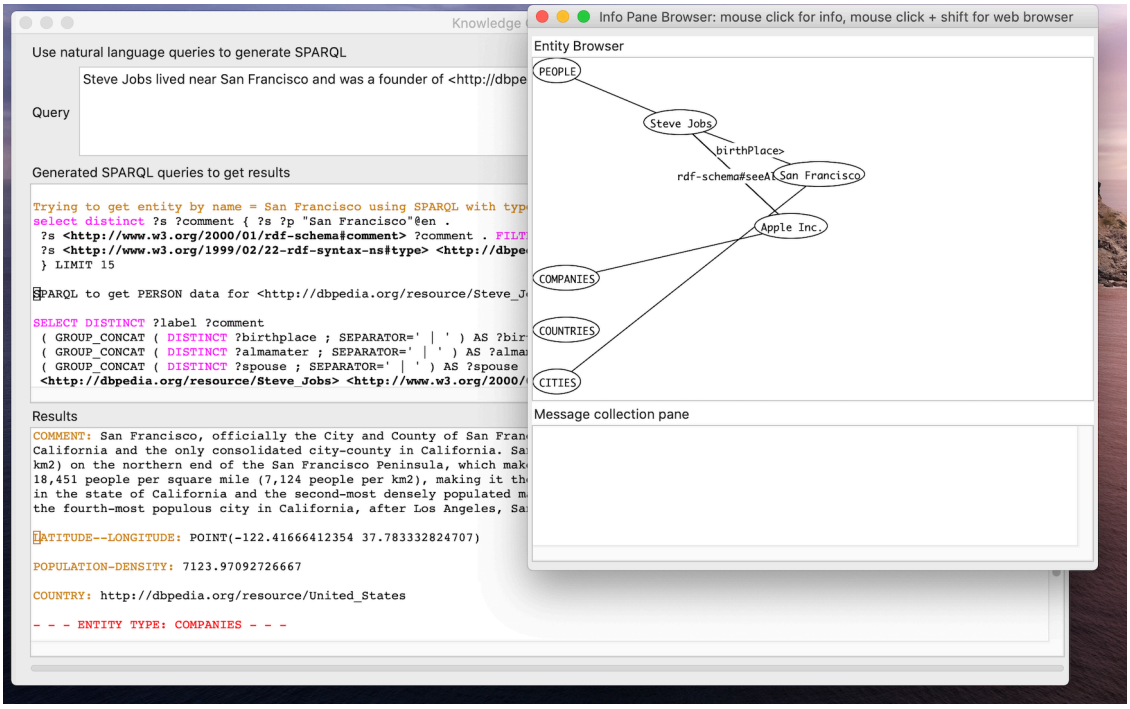
Also, this example is set up for people and companies. I may expand it in the future to other types of entities as I need them.

This example program takes several minutes to run since many SPARQL queries are made to DBPedia. I am a non-corporate member of the DBPedia organization. Here is a membership application[81] if you are interested in joining me there.

[81] https://www.dbpedia.org/members/membership/

# Knowledge Graph Navigator

The Knowledge Graph Navigator (which I will often refer to as KGN) is a tool for processing a set of entity names and automatically exploring the public Knowledge Graph DBPedia[82] using SPARQL queries. I started to write KGN for my own use, to automate some things I used to do manually when exploring Knowledge Graphs, and later thought that KGN might be useful also for educational purposes. KGN shows the user the auto-generated SPARQL queries so hopefully the user will learn by seeing examples. KGN uses NLP code developed in earlier chapters and we will reuse that code with a short review of using the APIs.




**UI for the Knowledge Graph Navigator**

After looking at generated SPARQL for an example query use of the application, we will start a process of bottom up development, first writing low level functions to automate SPARQL queries, writing utilities we will need for the UI, and finally writing the UI. Some of the problems we will need to solve along the way will be colorizing the output the user sees in the UI and implementing a progress bar so the application user does not think the application is "hanging" while generating and making SPARQL queries to DBPedia.

Since the DBPedia queries are time consuming, we will also implement a caching layer using SQLite that will make the app more responsive. The cache is especially helpful during development when

[82] http://dbpedia.org

the same queries are repeatedly used for testing.

The code for this application is in the directory **src/kgn**. KGN is a long example application for a book and we will not go over all of the code. Rather, I hope to provide you with a roadmap overview of the code, diving in on code that you might want to reuse for your own projects and some representative code for generating SPARQL queries.

# Example Output

Before we get started studying the implementation, let's look at sample output in order to help give meaning to the code we will look at later. Consider a query that a user might type into the top query field in the KGN app:

```
1     Steve Jobs lived near San Francisco and was
2     a founder of \<http://dbpedia.org/resource/Apple_Inc.\>
```

The system will try to recognize entities in a query. If you know the DBPedia URI of an entity, like the company Apple in this example, you can use that directly. Note that in the SPARQL URIs are surrounded with angle bracket characters.

The application prints out automatically generated SPARQL queries. For the above listed example query the following output will be generated (some editing to fit page width):

```
Trying to get entity by name = Steve Jobs using SPARQL with type:


select distinct ?s ?comment { ?s ?p "Steve Jobs"@en .
 ?s <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
   FILTER ( lang ( ?comment ) = 'en' ) .
 ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://dbpedia.org/ontology/Person> .
 } LIMIT 15


Trying to get entity by name = San Francisco using SPARQL with type:
```

```
select distinct ?s ?comment { ?s ?p "San Francisco"@en .
 ?s <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
   FILTER ( lang ( ?comment ) = 'en' ) .
 ?s <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://dbpedia.org/ontology/City> .
 } LIMIT 15
```

SPARQL to get PERSON data for <http://dbpedia.org/resource/Steve_Jobs>:

```
SELECT DISTINCT ?label ?comment
 ( GROUP_CONCAT ( DISTINCT ?birthplace ; SEPARATOR=' | ' ) AS ?birthplace )
 ( GROUP_CONCAT ( DISTINCT ?almamater ; SEPARATOR=' | ' ) AS ?almamater )
 ( GROUP_CONCAT ( DISTINCT ?spouse ; SEPARATOR=' | ' ) AS ?spouse ) {
 <http://dbpedia.org/resource/Steve_Jobs>
   <http://www.w3.org/2000/01/rdf-schema#comment>
   ?comment .
 FILTER ( lang ( ?comment ) = 'en' ) .
 OPTIONAL { <http://dbpedia.org/resource/Steve_Jobs>
            <http://dbpedia.org/ontology/birthPlace>
            ?birthplace } .
 OPTIONAL { <http://dbpedia.org/resource/Steve_Jobs>
            <http://dbpedia.org/ontology/almaMater>
            ?almamater } .
 OPTIONAL { <http://dbpedia.org/resource/Steve_Jobs>
            <http://dbpedia.org/ontology/spouse>
            ?spouse } .
 OPTIONAL { <http://dbpedia.org/resource/Steve_Jobs>
            <http://www.w3.org/2000/01/rdf-schema#label>
            ?label .
 FILTER ( lang ( ?label ) = 'en' ) }
 } LIMIT 10
```

SPARQL to get CITY data for <http://dbpedia.org/resource/San_Francisco>:

```
SELECT DISTINCT ?label ?comment
 ( GROUP_CONCAT ( DISTINCT ?latitude_longitude ; SEPARATOR=' | ' )
     AS ?latitude_longitude )
 ( GROUP_CONCAT ( DISTINCT ?populationDensity ; SEPARATOR=' | ' )
     AS ?populationDensity )
 ( GROUP_CONCAT ( DISTINCT ?country ; SEPARATOR=' | ' )
     AS ?country ) {
 <http://dbpedia.org/resource/San_Francisco>
   <http://www.w3.org/2000/01/rdf-schema#comment>
   ?comment .
      FILTER ( lang ( ?comment ) = 'en' ) .
 OPTIONAL { <http://dbpedia.org/resource/San_Francisco>
            <http://www.w3.org/2003/01/geo/wgs84_pos#geometry>
            ?latitude_longitude } .
 OPTIONAL { <http://dbpedia.org/resource/San_Francisco>
            <http://dbpedia.org/ontology/PopulatedPlace/populationDensity>
            ?populationDensity } .
 OPTIONAL { <http://dbpedia.org/resource/San_Francisco>
            <http://dbpedia.org/ontology/country>
            ?country } .
 OPTIONAL { <http://dbpedia.org/resource/San_Francisco>
            <http://www.w3.org/2000/01/rdf-schema#label>
            ?label . }
 } LIMIT 30
```

SPARQL to get COMPANY data for <http://dbpedia.org/resource/Apple_Inc.>:

```
SELECT DISTINCT ?label ?comment ( GROUP_CONCAT ( DISTINCT ?industry ; SEPARATOR=' | \
' )
      AS ?industry )
 ( GROUP_CONCAT ( DISTINCT ?netIncome ; SEPARATOR=' | ' )
      AS ?netIncome )
 ( GROUP_CONCAT ( DISTINCT ?numberOfEmployees ; SEPARATOR=' | ' )
      AS ?numberOfEmployees ) {
 <http://dbpedia.org/resource/Apple_Inc.>
    <http://www.w3.org/2000/01/rdf-schema#comment> ?comment .
 FILTER ( lang ( ?comment ) = 'en' ) .
 OPTIONAL { <http://dbpedia.org/resource/Apple_Inc.>
            <http://dbpedia.org/ontology/industry>
            ?industry } .
 OPTIONAL { <http://dbpedia.org/resource/Apple_Inc.>
```

```
            <http://dbpedia.org/ontology/netIncome> ?netIncome } .
 OPTIONAL { <http://dbpedia.org/resource/Apple_Inc.>
            <http://dbpedia.org/ontology/numberOfEmployees> ?numberOfEmployees } .
 OPTIONAL { <http://dbpedia.org/resource/Apple_Inc.>
            <http://www.w3.org/2000/01/rdf-schema#label> ?label .
              FILTER ( lang ( ?label ) = 'en' ) }
 } LIMIT 30
```

```
DISCOVERED RELATIONSHIP LINKS:


<http://dbpedia.org/resource/Steve_Jobs>    ->
    <http://dbpedia.org/ontology/birthPlace>    ->
    <http://dbpedia.org/resource/San_Francisco>
<http://dbpedia.org/resource/Steve_Jobs>    ->
    <http://dbpedia.org/ontology/occupation>    ->
    <http://dbpedia.org/resource/Apple_Inc.>
<http://dbpedia.org/resource/Steve_Jobs>    ->
    <http://dbpedia.org/ontology/board>         ->
    <http://dbpedia.org/resource/Apple_Inc.>
<http://dbpedia.org/resource/Steve_Jobs>    ->
    <http://www.w3.org/2000/01/rdf-schema#seeAlso> ->
    <http://dbpedia.org/resource/Apple_Inc.>
<http://dbpedia.org/resource/Apple_Inc.>    ->
    <http://dbpedia.org/property/founders>      ->
    <http://dbpedia.org/resource/Steve_Jobs>
```

After listing the generated SPARQL for finding information for the entities in the query, KGN searches for relationships between these entities. These discovered relationships can be seen at the end of the last listing. Please note that this step makes SPARQL queries on **O(n^2)** where **n** is the number of entities. Local caching of SPARQL queries to DBPedia helps make processing several entities possible.

In addition to showing generated SPARQL and discovered relationships in the middle text pane of the application, KGN also generates formatted results that are also displayed in the bottom text pane:

- - - ENTITY TYPE: PEOPLE - - -

LABEL: Steve Jobs

COMMENT: Steven Paul "Steve" Jobs was an American information technology entrepreneur and inventor. He was the co-founder, chairman, and chief executive officer (CEO) of Apple Inc.; CEO and majority shareholder of Pixar Animation Studios; a member of The Walt Disney Company's board of directors following its acquisition of Pixar; and founder, chairman, and CEO of NeXT Inc. Jobs is widely recognized as a pioneer of the microcomputer revolution of the 1970s and 1980s, along with Apple co-founder Steve Wozniak. Shortly after his death, Jobs's official biographer, Walter Isaacson, described him as a "creative entrepreneur whose passion for perfection and ferocious drive revolutionized six industries: personal computers, animated movies, music, phones

BIRTHPLACE: http://dbpedia.org/resource/San_Francisco

ALMAMATER: http://dbpedia.org/resource/Reed_College

SPOUSE: http://dbpedia.org/resource/Laurene_Powell_Jobs

- - - ENTITY TYPE: CITIES - - -

LABEL:  San Francisco

COMMENT: San Francisco, officially the City and County of San Francisco, is the cultural, commercial, and financial center of Northern California and the only consolidated city-county in California. San Francisco encompasses a land area of about 46.9 square miles (121 km2) on the northern end of the San Francisco Peninsula, which makes it the smallest county in the state. It has a density of about 18,451 people per square mile (7,124 people per km2), making it the most densely settled large city (population greater than 200,000) in the state of California and the second-most densely populated major city in the United States after New York City. San Francisco is the fourth-most populous city in California, after Los Angeles, San Diego, and San Jose, and the 13th-most populous cit

LATITUDE--LONGITUDE: POINT(-122.41666412354 37.783332824707)

POPULATION-DENSITY: 7123.97092726667

COUNTRY: http://dbpedia.org/resource/United_States

```
- - - ENTITY TYPE: COMPANIES - - -

LABEL: Apple Inc.

COMMENT: Apple Inc. is an American multinational technology company headquartered
in Cupertino,
California, that designs, develops, and sells consumer electronics,
computer software, and online services. Its hardware products include the
iPhone smartphone, the iPad tablet computer, the Mac personal computer, the
iPod portable media player, the Apple Watch smartwatch, and the Apple TV digital
media player. Apple's consumer software includes the macOS and iOS operating
systems, the iTunes media player, the Safari web browser, and the iLife and
iWork creativity and productivity suites. Its online services include the
iTunes Store, the iOS App Store and Mac App Store, Apple Music, and iCloud.

INDUSTRY: http://dbpedia.org/resource/Computer_hardware |
          http://dbpedia.org/resource/Computer_software |
          http://dbpedia.org/resource/Consumer_electronics |
          http://dbpedia.org/resource/Corporate_Venture_Capital |
          http://dbpedia.org/resource/Digital_distribution |
          http://dbpedia.org/resource/Fabless_manufacturing

NET-INCOME: 5.3394E10

NUMBER-OF-EMPLOYEES: 115000
```

Hopefully after reading through sample output and seeing the screen shot of the application, you now have a better idea what this example application does. Now we will look at project configuration and then implementation.

# Project Configuration and Running the Application

The following listing of **kgn.asd** shows the ten packages this example depends on (five of these are also examples in this book, and five are in the public Quicklisp repository):

```
;;;; knowledgegraphnavigator.asd

(asdf:defsystem #:kgn
  :description "Describe dbpedia here"
  :author "Mark Watson <markw@markwatson.com>"
  :license "Apache 2"
  :depends-on (#:sqlite #:cl-json #:alexandria #:drakma #:myutils #:lw-grapher
               #:trivial-open-browser #:entities #:entity-uris #:kbnlp)
  :components ((:file "package")
               (:file "ui-text")
               (:file "utils")
               (:file "sparql")
               (:file "colorize")
               (:file "user-interface")
               (:file "option-pane")
               (:file "kgn")
               (:file "gui")
               (:file "nlp")
               (:file "sparql-results-to-english")
               (:file "gen-output")))
```

You are probably aware of many of the dependency libraries used here but you may not have seen **trivial-open-browser** which we will use to open a web browser to URIs for human readable information on DBPedia.

Listing of **package.lisp**:

```
;;;; package.lisp

(defpackage #:kgn
  (:use #:cl #:alexandria #:myutils #:sqlite #:myutils
   #:lw-grapher #:trivial-open-browser #:entities #:entity-uris
   #:kbnlp #:CAPI)
  (:export #:kgn))
```

The free personal edition of LispWorks does not support initialization files so you must manually load Quicklisp from the Listener Window when you first start LispWorks Personal as seen in the following repl listing (edited to remove some output for brevity). Once Quicklisp is loaded we then use **ql:quickload** to load the example in this chapter (some output removed for brevity):

```
CL-USER 1 > (load "~/quicklisp/setup.lisp")
; Loading text file /Users/markw/quicklisp/setup.lisp
; Loading /Applications/LispWorks Personal 7.1/...
;; Creating system "COMM"
#P"/Users/markw/quicklisp/setup.lisp"

CL-USER 2 > (ql:quickload "kgn")
To load "kgn":
  Load 1 ASDF system:
    kgn
; Loading "kgn"
.
"Starting to load data...."
"....done loading data."
"#P\"/Users/markw/GITHUB/common-lisp/entity-uris/entity-uris.lisp\""
"current directory:"
"/Users/markw/GITHUB/common-lisp/entity-uris"
"Starting to load data...."
"....done loading data."
[package kgn]
To load "sqlite":
  Load 1 ASDF system:
    sqlite
; Loading "sqlite"
To load "cl-json":
  Load 1 ASDF system:
    cl-json
; Loading "cl-json"
To load "drakma":
  Load 1 ASDF system:
    drakma
; Loading "drakma"
.To load "entity-uris":
  Load 1 ASDF system:
    entity-uris
; Loading "entity-uris"
("kgn")
CL-USER 3 > (kgn:kgn)
#<KGN::KGN-INTERFACE "Knowledge Graph Navigator" 40201E91DB>
```

Please note that I assume that you have configured all of the examples for this book for discoverability by Quicklisp as per the section Setup for Local Quicklisp Projects in Appendix A.

When the KGN application starts a sample query is randomly chosen. Queries with many entities

can take a while to process, especially when you first start using this application. Every time KGN makes a web service call to DBPedia the query and response are cached in a SQLite database in ~/**.kgn_cache.db** which can greatly speed up the program, especially in development mode when testing a set of queries. This caching also takes some load off of the public DBPedia endpoint, which is a polite thing to do.

I use LispWorks Professional and add two utility functions to the bottom on my ~/**.lispworks** configuration file (you can't do this with LispWorks Personal):

```
;;; The following lines added by ql:add-to-init-file:
#-quicklisp
(let ((quicklisp-init
       (merge-pathnames
         "quicklisp/setup.lisp"
         (user-homedir-pathname))))
  (when (probe-file quicklisp-init)
    (load quicklisp-init)))

(defun ql (x) (ql:quickload x))
(defun qlp (x)
  (ql:quickload x)
  (SYSTEM::%IN-PACKAGE (string-upcase x) :NEW T))
```

Function **ql** is just a short alias to avoid frequently typing **ql:quickload** and **qlp** loads a Quicklisp project and then performs an **in-package** of the Common Lisp package with the same name as the Quicklisp project.

# Review of NLP Utilities Used in Application

Here is a quick review of NLP utilities we saw earlier:

- kbnlp:make-text-object
- kbnlp::text-human-names
- kbnlp::text-place-name
- entity-uris:find-entities-in-text
- entity-uris:pp-entities

The following code snippets show example calls to the relevant NLP functions and the generated output:

```
KGN 39 > (setf text "Bill Clinton went to Canada")
"Bill Clinton went to Canada"

KGN 40 > (setf txtobj (kbnlp:make-text-object text))
#S(TEXT :URL "" :TITLE "" :SUMMARY "<no summary>" :CATEGORY-TAGS (("computers_micros\
oft.txt" 0.00641) ("religion_islam.txt" 0.00357)) :KEY-WORDS NIL :KEY-PHRASES NIL :H\
UMAN-NAMES ("Bill Clinton") :PLACE-NAMES ("Canada") :COMPANY-NAMES NIL :TEXT #("Bill\
" "Clinton" "went" "to" "Canada") :TAGS #("NNP" "NNP" "VBD" "TO" "NNP"))

KGN 41 > (kbnlp::text-human-names txtobj)
("Bill Clinton")

KGN 42 >
(loop for key being the hash-keys of  (entity-uris:find-entities-in-text text)
  using (hash-value value)
  do (format t "key: ~S value: ~S~%" key value))
key: "people" value: (("Bill Clinton" "<http://dbpedia.org/resource/Bill_Clinton>"))
key: "countries" value: (("Canada" "<http://dbpedia.org/resource/Canada>"))
NIL
```

The code using **loop** at the end of the last repl listing that prints keys and values of a hash table is from the Common Lisp Cookbook web site[83] in the section "Traversing a Hash Table."

## Developing Low-Level SPARQL Utilities

I use the standard command line **curl** utility program with the Common Lisp package **uiop** to make HTML GET requests to the DBPedia public Knowledge Graph and the package **drakma** to url-encode parts of a query. The source code is in **src/kgn/sparql.lisp**. In lines 8, 24, 39, and 55 I use some caching code that we will look at later. The nested **replace-all** statements in lines 12-13 are a kluge to remove Unicode characters that occasionally caused runtime errors in the KGN application.

```
1  (in-package #:kgn)
2
3  (ql:quickload "cl-json")
4  (ql:quickload "drakma")
5
6  (defun sparql-dbpedia (query)
7    (let* (ret
8           (cr (fetch-result-dbpedia query))
9           (response
```

[83] http://cl-cookbook.sourceforge.net/hashes.html

```
          (or
           cr
           (replace-all
            (replace-all
             (uiop:run-program
              (list
               "curl"
               (concatenate 'string
                            "https://dbpedia.org/sparql?query="
                            (drakma:url-encode query :utf-8)
                            "&format=json"))
              :output :string)
             "\\u2013" " ")
            "\\u" " "))))
    (save-query-result-dbpedia query response)
    (ignore-errors
      (with-input-from-string
          (s response)
        (let ((json-as-list (json:decode-json s)))
          (setf
           ret
           (mapcar #'(lambda (x)
                       ;;(pprint x)
                       (mapcar #'(lambda (y)
                                   (list (car y) (cdr (assoc :value (cdr y))))) x))
                   (cdr (cadddr (cadr json-as-list)))))))))
    ret))

(defun sparql-ask-dbpedia (query)
  (let* ((cr (fetch-result-dbpedia query))
         (response
          (or
           cr
           (replace-all
            (replace-all
             (uiop:run-program
              (list
               "curl"
               (concatenate 'string
                            "https://dbpedia.org/sparql?query="
                            (drakma:url-encode query :utf-8)
                            "&format=json"))
              :output :string)
```

```
53              "\\u2013" " ")
54             "\\u" " "))))
55     (save-query-result-dbpedia query response)
56     (if  (search "true" response)
57         t
58       nil)))
```

The code for replacing Unicode characters is messy but prevents problems later when we are using the query results in the example application.

The code **(json-as-list (json:decode-json s))** on line 28 converts a deeply nested JSON response to nested Common Lisp lists. You may want to print out the list to better understand the **mapcar** expression on lines 31-35. There is no magic to writing expressions like this, in a repl I set **json-as-list** to the results of one query, and I spent a minute or two experimenting with the nested **mapcar** expression to get it to work with my test case.

The implementation for **sparql-ask-dbpedia** in lines 38-58 is simpler because we don't have to fully parse the returned SPARQL query results. A SPARQL **ask** type query returns a true/false answer to a query. We will use this to determine the types of entities in query text. While our NLP library identifies entity types, making additional **ask** queries to DBPedia to verify entity types will provide better automated results.

# Implementing the Caching Layer

While developing KGN and also using it as an end user, many SPARQL queries to DBPedia contain repeated entity names so it makes sense to write a caching layer. We use a SQLite database "~/.kgn_-cache.db" to store queries and responses.

The caching layer is implemented in the file **kgn/utils.lisp** and some of the relevant code is listed here:

```
1  ;;; SqList caching for SPARQL queries:
2
3  (defvar *db-path* (pathname "~/.kgn_cache.db"))
4
5  (defun create-dbpedia ()
6    (sqlite:with-open-database (d *db-path*)
7      (ignore-errors
8        (sqlite:execute-single d
9          "CREATE TABLE dbpedia (query string  PRIMARY KEY ASC, result string)"))))
10
11 (defun save-query-result-dbpedia (query result)
12   (sqlite:with-open-database (d *db-path*)
```

```
    (ignore-errors
      (sqlite:execute-to-list d
                       "insert into dbpedia (query, result) values (?, ?)"
                       query result))))
(defun fetch-result-dbpedia (query)
  (sqlite:with-open-database (d *db-path*)
    (cadar
     (sqlite:execute-to-list d
                      "select * from dbpedia where query = ?" query))))
```

This caching layer greatly speeds up my own personal use of KGN. Without caching, queries that contain many entity references simply take too long to run. The UI for the KGN application has a menu option for clearing the local cache but I almost never use this option because growing a large cache that is tailored for the types of information I search for makes the entire system much more responsive.

## Utilities to Colorize SPARQL and Generated Output

When I first had the basic functionality of KGN working, I was disappointed by how the application looked as all black text on a white background. Every editor and IDE I use colorizes text in an appropriate way so I took advantage of the function **capi::write-string-with-properties** to (fairly) easily implement color hilting SPARQL queries.

The code in the following listing is in the file **kgn/colorize.lisp**. When I generate SPARQL queries to show the user I use the characters "@@" as placeholders for end of lines in the generated output. In line 5 I am ensuring that there are spaces around these characters so they get tokenized properly. In the loop starting at line 7 I process the tokens checking each one to see if it should have a color associated with it when it is written to the output stream.

```
1  (in-package #:kgn)
2
3  (defun colorize-sparql (s  &key (stream nil))
4    (let ((tokens (tokenize-string-keep-uri
5                       (replace-all s "@@" " @@ ")))
6          in-var)
7      (dolist (token tokens)
8        (if (> (length token) 0)
9            (if (or in-var (equal token "?"))
10               (capi::write-string-with-properties
11                 token
12                 '(:highlight :compiler-warning-highlight)
13                 stream)
```

```
            (if (find token '("where" "select" "distinct" "option" "filter"
                              "FILTER" "OPTION" "DISTINCT"
                              "SELECT" "WHERE")
                        :test #'equal)
                (capi::write-string-with-properties
                  token
                  '(:highlight :compiler-note-highlight)
                  stream)
              (if (equal (subseq token 0 1) "<")
                  (capi::write-string-with-properties
                    token
                    '(:highlight :bold)
                    stream)
                (if (equal token "@@")
                    (terpri stream)
                  (if (not (equal token "~")) (write-string token stream)))))))
      (if (equal token "?")
          (setf in-var t)
        (setf in-var nil))
      (if (and
           (not in-var)
           (not (equal token "?")))
          (write-string " " stream)))
    (terpri stream)))
```

Here is an example call to function **colorize-sparql**:

```
KGN 25 > (colorize-sparql "select ?s ?p  where {@@  ?s ?p \"Microsoft\" } @@  FILTER\
 (lang(?comment) = 'en')")
select ?s ?p where {
 ?s ?p "Microsoft" }
 FILTER ( lang ( ?comment ) = 'en' )
```

# Text Utilities for Queries and Results

The utilities in the file **kgn/ui-text.lisp** contain no CAPI UI code but are used by the CAPI UI code. The function **display-entity-results** is passed an output stream that during repl development is passed as **t** to get output in the repl and in the application will be the output stream attached to a text pane. The argument **r-list** is a list of results where each result is a list containing a result title and a list of key/value pairs:

```
(defun display-entity-results (output-stream r-list)
  (dolist (r r-list)
    (format output-stream "~%~%entity result:~%~S~%" r)
    (dolist (val r)
      (if (> (length (second val)) 0)
          (format output-stream "~%~a: ~a~%" (first val) (second val))))))

(defun get-URIs-in-query (query) ;; URIs contain < > brackets
  (let (ret
        w
        (ll (coerce query 'list))
        in-uri)
    (dolist (ch ll)
      (if in-uri
          (if (equal ch #\>)
              (setf w (cons ch w)
                    ret (cons (coerce (reverse w) 'string) ret)
                    in-uri nil
                    w nil)
            (setf w (cons ch w))))
      (if (equal ch #\<) (setf in-uri t
                               w (cons #\< w))))
    ret))
```

The function **get-URIs-in-query** in lines 8-23 simply looks for URIs and saves them in a list.

In SPARQL queries, URIs are surround by angle brackets. The following code remove the brackets and embedded URIs. The function **remove-uris-from-query** simply looks for URIs in an input string and removes them:

```
(defun remove-uris-from-query (query) ;; URIs contain < > brackets
  (let (ret
        (ll (coerce query 'list))
        in-uri)
    (dolist (ch ll)
      (if (equal ch #\<) (setf in-uri t))
      (if (not in-uri)
          (setf ret (cons ch ret)))
      (if (equal ch #\>) (setf in-uri nil)))
    (coerce (reverse ret) 'string)))
```

Here is a test:

```
KGN 26 >
(remove-uris-from-query
 "<http://dbpedia.org/resource/Bill_Gates> visited <http://dbpedia.org/resource/Appl\
e_Inc.>")
" visited "
```

Given a list of URIs, the following function makes multiple SPARQL queries to DBPedia to get more information using the function **get-name-and-description-for-uri** that we will look at later:

```
1 (defun handle-URIs-in-query (query)
2   (let* ((uris (get-URIs-in-query query))
3          (entity-names (map 'list #'get-name-and-description-for-uri uris)))
4     (mapcar #'list uris (map 'list #'second entity-names))))
```

The following repl show a call to **handle-URIs-in-query**:

```
KGN 30 > (pprint (handle-URIs-in-query "<http://dbpedia.org/resource/Bill_Gates> vis\
ited <http://dbpedia.org/resource/Apple_Inc.>"))

(("<http://dbpedia.org/resource/Apple_Inc.>"
  "Apple Inc. is an American multinational technology company headquartered in Cuper\
tino, California, that designs, develops, and sells consumer electronics, computer s\
oftware, and online services. Its hardware products include the iPhone smartphone, t\
he iPad tablet computer, the Mac personal computer, the iPod portable media player, \
the Apple Watch smartwatch, and the Apple TV digital media player. Apple's consumer \
software includes the macOS and iOS operating systems, the iTunes media player, the \
Safari web browser, and the iLife and iWork creativity and productivity suites. Its \
online services include the iTunes Store, the iOS App Store and Mac App Store, Apple\
 Music, and iCloud.")
 ("<http://dbpedia.org/resource/Bill_Gates>"
  "William Henry \"Bill\" Gates III (born October 28, 1955) is an American business \
magnate, investor, author and philanthropist. In 1975, Gates and Paul Allen co-found\
ed Microsoft, which became the world's largest PC software company. During his caree\
r at Microsoft, Gates held the positions of chairman, CEO and chief software archite\
ct, and was the largest individual shareholder until May 2014. Gates has authored an\
d co-authored several books."))
```

The function **get-entity-data-helper** processes the user's query and finds entities using both the NLP utilities from earlier in this book and by using SPARQL queries to DBPedia. Something new are calls to the function **updater** (lines 10-13, 17-20, and 29-31) that is defined as an optional argument. As we will see later, we will pass in a function value in the application that updates the progress bar at the bottom of the application window.

```
(defun get-entity-data-helper (original-query
                               &key
                               (message-stream t)
                               (updater nil))
  (let* ((uri-data (handle-URIs-in-query original-query))
         (query (remove-uris-from-query original-query))
         ret
         (el (entities:text->entities query))
         (people (entities:entities-people el)))
    (if updater
        (let ()
          (setf *percent* (+ *percent* 2))
          (funcall updater *percent*)))
    (let* ((companies (entities:entities-companies el))
           (countries (entities:entities-countries el))
           (cities (entities:entities-cities el)))
      (if updater
          (let ()
            (setf *percent* (+ *percent* 2))
            (funcall updater *percent*)))
      (let* ((products (entities:entities-products el))
             places
             companies-uri people-uri countries-uri cities-uri places-uri
             (text-object (kbnlp:make-text-object query))
             (to-place-names (kbnlp::text-place-names text-object))
             (to-people (kbnlp::text-human-names text-object)))

        (if updater
          (let ()
            (setf *percent* (+ *percent* 3))
            (funcall updater *percent*)))

        (dolist (ud uri-data)
          (if (ask-is-type-of (first ud) "<http://dbpedia.org/ontology/Company>")
              (setf companies-uri (cons ud companies-uri)))
          (if (ask-is-type-of (first ud) "<http://dbpedia.org/ontology/Person>")
              (setf people-uri (cons ud people-uri)))
          (if (ask-is-type-of (first ud) "<http://dbpedia.org/ontology/Country>")
              (setf countries-uri (cons ud countries-uri)))
          (if (ask-is-type-of (first ud) "<http://dbpedia.org/ontology/City>")
              (setf cities-uri (cons ud cities-uri)))
          (if (ask-is-type-of (first ud) "<http://dbpedia.org/ontology/Place>")
              (setf places-uri (cons ud places-uri))))
```

```
        (dolist (place to-place-names)
          (if (and
               (not (member place countries :test #'equal))
               (not (member place cities :test #'equal)))
              (setf places (cons place places))))
         (dolist (person to-people)
           (if (not (member person people :test #'equal))
             (setf people (cons person people))))
         (let ((entity-list
                (list
                 (cons :people
                       (append
                        (loop for person in people collect
                              (dbpedia-get-entities-by-name
                               person
                               "<http://dbpedia.org/ontology/Person>"
                               "<http://schema.org/Person>"
                               :message-stream message-stream))
                        (list people-uri)))
                 (cons :countries
                       (append
                        (loop for country in countries collect
                              (dbpedia-get-entities-by-name
                               country
                               "<http://dbpedia.org/ontology/Country>"
                               "<http://schema.org/Country>"
                               :message-stream message-stream))
                        (list countries-uri)))
                 (cons :cities
                       (append
                        (loop for city in cities collect
                              (dbpedia-get-entities-by-name
                               city
                               "<http://dbpedia.org/ontology/City>"
                               "<http://schema.org/City>"
                               :message-stream message-stream))
                        (list cities-uri)))
                 (cons :places
                       (append
                        (loop for place in places collect
                              (dbpedia-get-entities-by-name
                               place
                               "<http://dbpedia.org/ontology/Place>"
```

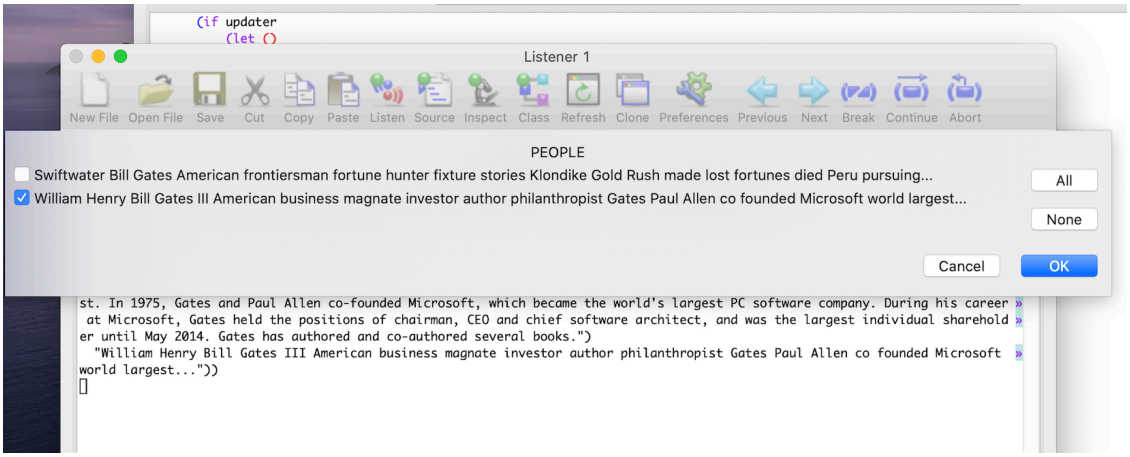
```
                                "<http://schema.org/Place>"
                                :message-stream message-stream))
                         (list places-uri)))
                  (cons :products
                        (loop for product in products collect
                              (dbpedia-get-entities-by-name
                               product
                               "<http://dbpedia.org/ontology/Product>"
                               "<http://schema.org/Product>"
                               :message-stream message-stream)))
                  (cons :companies
                        (append
                         (loop for company in companies collect
                               (dbpedia-get-entities-by-name
                                company
                                "<http://dbpedia.org/ontology/Organization>"
                                "<http://schema.org/Organization>"
                                :message-stream message-stream))
                         (list companies-uri))))))
            (setf ret (prompt-selection-list entity-list))
            (format t "~%~%--------- ret:~%~%~S~%~%" ret)
            ret)))))
```

This function presents a CAPI popup list selector to the user so the following listed output depends on which possible entities are selected in this list. If you run the following repl example, you will see a popup window that will ask you to verify discovered entities; the user needs to check all discovered entities that are relevant to their interests.

```
KGN 33 > (pprint (get-entity-data-helper "Bill Gates at Microsoft"))
((:PEOPLE
  (("<http://dbpedia.org/resource/Bill_Gates>"
    "William Henry \"Bill\" Gates III (born October 28, 1955) is an American busines\
s magnate, investor, author and philanthropist. In 1975, Gates and Paul Allen co-fou\
nded Microsoft, which became the world's largest PC software company. During his car\
eer at Microsoft, Gates held the positions of chairman, CEO and chief software archi\
tect, and was the largest individual shareholder until May 2014. Gates has authored \
and co-authored several books.")))
 (:COMPANIES
  (("<http://dbpedia.org/resource/Microsoft>"
    "Microsoft Corporation / 02C8ma 026Akr 0259 02CCs 0252ft, -ro 028A-, - 02CCs 025\
4 02D0ft/ (commonly referred to as Microsoft or MS) is an American multinational tec\
hnology company headquartered in Redmond, Washington, that develops, manufactures, l\
icenses, supports and sells computer software, consumer electronics and personal com\
```

```
puters and services. Its best known software products are the Microsoft Windows line\
 of operating systems, Microsoft Office office suite, and Internet Explorer and Edge\
 web browsers. Its flagship hardware products are the Xbox video game consoles and t\
he Microsoft Surface tablet lineup. As of 2011, it was the world's largest software \
maker by revenue, and one of the world's most valuable companies."))))
```

The popup list in the last example looks like:




**Popup list shows the user possible entity resolutions for each entity found in the input query. The user selects the resolved entities to use.**

In this example there were two "Bill Gates" entities, one an early American frontiersman, the other the founder of Microsoft and I chose the latter person to continue finding information about.

After identifying all of the entities that the user intended, the function **entity-results->relationship-link** in the following listing is called to make additional SPARQL queries to discover possible relationships between these entities. This function is defined in the file **ui-text.lisp**.

```
(defun entity-results->relationship-links (results
                  &key (message-stream t) (updater nil))
  (let (all-uris
        relationship-statements
        (sep " -> "))
    (dolist (r results)
      (dolist (entity-data (cdr r))
        (dolist (ed entity-data)
          (setf all-uris (cons (first ed) all-uris)))))
    (dolist (e1 all-uris)
      (dolist (e2 all-uris)
        (if updater
            (let ()
              (setf *percent* (+ *percent* 1))
```

```
              (funcall updater *percent*)))
        (if (not (equal e1 e2))
            (let ((l1 (dbpedia-get-relationships e1 e2))
                  (l2 (dbpedia-get-relationships e2 e1)))
              (dolist (x l1)
                (setf relationship-statements
                  (cons (list e1 e2 x) relationship-statements)))
              (dolist (x l2)
                (print (list "x l2:" x))
                (setf relationship-statements
                  (cons (list e2 e1 x) relationship-statements)))))))
    (setf relationship-statements
      (remove-duplicates relationship-statements :test #'equal))
    ;;(terpri message-stream)
    (capi::write-string-with-properties
      "DISCOVERED RELATIONSHIP LINKS:"
      '(:highlight :compiler-warning-highlight) message-stream)
    (terpri message-stream) (terpri message-stream)
    (dolist (rs relationship-statements)
      (format message-stream "~43A" (first rs))
      (capi::write-string-with-properties
        sep
        '(:highlight :compiler-warning-highlight) message-stream)
      (format message-stream "~43A" (third rs))
      (capi::write-string-with-properties
              sep
              '(:highlight :compiler-warning-highlight) message-stream)
      (format message-stream "~A" (second rs))
      (terpri message-stream))
    relationship-statements))
```

In the following repl listing we create some test data of the same form as we get from calling function **get-entity-data-helper** seen in a previous listing and try calling **entity-results->relationship-links** with this data:

```
KGN 36 > (setf results '((:PEOPLE
  (("<http://dbpedia.org/resource/Bill_Gates>"
    "William Henry \"Bill\" Gates III (born October 28, 1955) is an American busines\
s magnate, investor, author and philanthropist. In 1975, Gates and Paul Allen co-fou\
nded Microsoft, which became the world's largest PC software company. During his car\
eer at Microsoft, Gates held the positions of chairman, CEO and chief software archi\
tect, and was the largest individual shareholder until May 2014. Gates has authored \
and co-authored several books.")))
 (:COMPANIES
  (("<http://dbpedia.org/resource/Microsoft>"
    "Microsoft Corporation / 02C8ma 026Akr 0259 02CCs 0252ft, -ro 028A-, - 02CCs 025\
4 02D0ft/ (commonly referred to as Microsoft or MS) is an American multinational tec\
hnology company headquartered in Redmond, Washington, that develops, manufactures, l\
icenses, supports and sells computer software, consumer electronics and personal com\
puters and services. Its best known software products are the Microsoft Windows line\
 of operating systems, Microsoft Office office suite, and Internet Explorer and Edge\
 web browsers. Its flagship hardware products are the Xbox video game consoles and t\
he Microsoft Surface tablet lineup. As of 2011, it was the world's largest software \
maker by revenue, and one of the world's most valuable companies.")))))
KGN 37 > (pprint (entity-results->relationship-links results))
(("<http://dbpedia.org/resource/Bill_Gates>"
  "<http://dbpedia.org/resource/Microsoft>"
  "<http://dbpedia.org/ontology/board>")
 ("<http://dbpedia.org/resource/Microsoft>"
  "<http://dbpedia.org/resource/Bill_Gates>"
  "<http://dbpedia.org/property/founders>")
 ("<http://dbpedia.org/resource/Microsoft>"
  "<http://dbpedia.org/resource/Bill_Gates>"
  "<http://dbpedia.org/ontology/keyPerson>"))
```

# Using LispWorks CAPI UI Toolkit

You can use the free LispWorks Personal Edition for running KGN. Using other Common Lisp implementations like Clozure-CL and SBCL will not work because the CAPI user interface library is proprietary to LispWorks. I would like to direct you to three online resources for learning CAPI:

- [LispWorks' main web age introducing CAPI[84]
- LispWorks' comprehensive CAPI documentation[85] for LispWorks version 7.1
- An older web site (last updated in 2011 but I find it useful for ideas): CAPI Cookbook[86]

[84] http://www.lispworks.com/products/capi.html
[85] http://www.lispworks.com/products/capi.html
[86] http://capi.plasticki.com/show?O4

I am not going to spend too much time in this chapter explaining my CAPI-based code. If you use LispWorks (either the free Personal or the Professional editions) you are likely to use CAPI and spending time on the official documentation and especially the included example programs is strongly recommended.

In the next section I will review the KGN specific application parts of the CAPI-based UI.

# Writing Utilities for the UI

The CAPI user interface code is in the file **src/kgn/gui.lisp** with some UI code in **options-pane.lisp** and **kgn.lisp**.

When printing results in the bottom Results Pane of the KGN application, I like to highlight the first line of each result using this function (first function in **kgn.lisp**):

```
1 (defun pprint-results (results &key (stream t))
2   (dolist (result (car results))
3     (terpri stream)
4     (capi::write-string-with-properties
5      (format nil  "~A:" (first result))
6      '(:highlight :compiler-warning-highlight) stream)
7     (format stream " ~A~%" (second result))))
```

I default the value for the input named variable **stream** to **t** so during development in a repl the output of this function goes to standard output. In the KGN app, I get an output stream for the bottom results pane in the user interface and pass that as the value for **stream** so output is directly written to the results pane.

CAPI allows you to define your own text highlight values. I use built-in ones like :**compiler-warning-highlight** that are always available to CAPI applications.

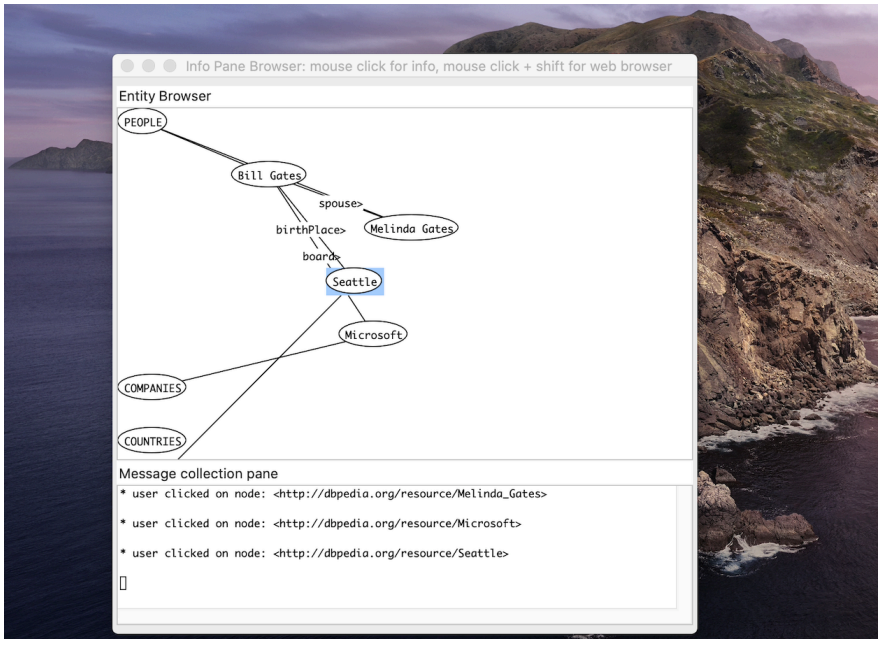
The file **kgn.lisp** defines several other utility functions including a utility that makes multiple SPARQL queries to get a name and description of an entity URI that removes end of line markers "@@" from a SPARQL query for fetching entity data, makes the query and extracts results for display:

```
(defun get-name-and-description-for-uri (uri)
  (let* ((sparql
          (replace-all
           (format nil "select distinct ?name ?comment { @@ ~
                        values ?nameProperty {<http://www.w3.org/2000/01/rdf-schema\
#label> <http://xmlns.com/foaf/0.1/name> } . @@ ~
                        ~A ?nameProperty ?name . @@ ~
                        ~A <http://www.w3.org/2000/01/rdf-schema#comment>  ?comment\
  . FILTER  (lang(?comment) = 'en') . @@ ~
                      } LIMIT 1" uri uri)
           "@@" " "))
         (results (sparql-dbpedia sparql)))
    (list
      (second (assoc :name (car results)))
      (second (assoc :comment (car results))))))
```

There are several other SPARQL query utility functions in the file **kgn.lisp** that I will not discuss but they follow a similar pattern of using specific SPARQL queries to fetch information from DBPedia.

At the top of the file **gui.lisp** I set three parameters for the width of the application window and a global flag used to toggle on and off showing the info-pane-grapher that you saw in the screen shot at the beginning of this chapter and that is also shown below:

```
(defvar *width* 1370)
(defvar *best-width* 1020)
(defvar *show-info-pane* t)
```




**UI for info-pane-grapher**

Since I just mentioned the **info-pane-grapher** this is a good time to digress to its implementation. This is in a different package and you will find the source code in **src/lw-grapher/info-pane-grapher.lisp**. I used the graph layout algorithm from ISI-Grapher Manual (by Gabriel Robbins)[87]. There is another utility in **src/lw-grapher/lw-grapher.lisp** that also displays a graph without mouse support and an attached information pane that is not used here but you might prefer it for reuse in your projects if you don't need mouse interactions.

The graph nodes are derived from the class **capi:pinboard-object**:

```
(defclass text-node (capi:pinboard-object)
  ((text :initarg :text :reader text-node-text)
   (string-x-offset :accessor text-node-string-x-offset)
   (string-y-offset :accessor text-node-string-y-offset)))
```

I customized how my graph nodes are drawn in a graph pane (this is derived from LispWorks example code):

[87] http://www.cs.virginia.edu/~robins/papers/The_ISI_Grapher_Manual.pdf

```
(defmethod capi:draw-pinboard-object (pinboard (self text-node)
                                               &key &allow-other-keys)
  (multiple-value-bind (X Y  width height)
      (capi:static-layout-child-geometry self)
    (let* ((half-width  (floor (1- width)  2))
           (half-height (floor (1- height) 2))
           (circle-x (+ X half-width))
           (circle-y (+ Y half-height))
           (background :white)
           (foreground (if background
                           :black
                         (capi:simple-pane-foreground pinboard)))
           (text (text-node-text self)))
      (gp:draw-ellipse pinboard
                       circle-x circle-y
                       half-width half-height
                       :filled t
                       :foreground background)
      (gp:draw-ellipse pinboard
                       circle-x circle-y
                       half-width half-height
                       :foreground foreground)
      (gp:draw-string pinboard
                      text
                      (+ X (text-node-string-x-offset self))
                      (+ Y (text-node-string-y-offset self))
                      :foreground foreground))))
```

Most of the work is done in the graph layout method that uses Gabriel Robbins' algorithm. Here I just show the signature and we won't go into implementation. If you are interested in modifying the layout code, I include a screen shot from ISI-Grapher manual showing the algorithm in a single page, see the file **src/lw-grapher/Algorithm from ISI-Grapher Manual.png**.

The following code snippets shows the method signature for the layout algorithm function in the file **src/lw-grapher/grapher.lisp**. I also include the call to **capi:graph-pane-nodes** that is the CLOS reader method for getting the list of node objects in a graph pane:

```
(defun graph-layout (self &key force)
  (declare (ignore force))
  (let* ((nodes (capi:graph-pane-nodes self))
    ...
```

The CAPI graph node model uses a function that is passed a node object and returns a list this node's

child node objects. There are several examples of this in the CAPI graph examples that are included with LispWorks (see the CAPI documentation).

In **src/lw-grapher/lw-grapher.lisp** I wrote a function that builds a graph layout and instead of passing in a "return children" function I found it more convenient to wrap this process, accepting a list of graph nodes and graph edges as function arguments:

```
(in-package :lw-grapher)

;; A Grapher (using the layout algorithm from the ISI-Grapher
;; user guide) with an info panel

(defun make-info-panel-grapher (h-root-name-list h-edge-list
                                h-callback-function-click
                                h-callback-function-shift-click)
  (let (edges roots last-selected-node node-callback-click
          node-callback-click-shift output-pane)
    (labels
        ((handle-mouse-click-on-pane (pane x y)
           (ignore-errors
             (let ((object (capi:pinboard-object-at-position pane x y)))
               (if object
                   (let ()
                     (if last-selected-node
                         (capi:unhighlight-pinboard-object pane
                             last-selected-node))
                     (setf last-selected-node object)
                     (capi:highlight-pinboard-object pane object)
                     (let ((c-stream (collector-pane-stream output-pane)))
                       (format c-stream
                         (funcall node-callback-click
                           (text-node-full-text object)))
                       (terpri c-stream)))))))
         (handle-mouse-click-shift-on-pane (pane x y)
           (ignore-errors
             (let ((object
                    (capi:pinboard-object-at-position pane x y)))
               (if object
                   (let ()
                     (if last-selected-node
                         (capi:unhighlight-pinboard-object
                           pane last-selected-node))
                     (setf last-selected-node object)
```

```
                    (capi:highlight-pinboard-object pane object)
                    (let ((c-stream
                           (collector-pane-stream output-pane)))
                      (format c-stream
                        (funcall node-callback-click-shift
                          (text-node-full-text object)))
                      (terpri c-stream)))))))

(info-panel-node-children-helper (node-text)
  (let (ret)
    (dolist (e edges)
      (if (equal (first e) node-text)
          (setf ret (cons (second e) ret))))
    (reverse ret)))

(make-info-panel-grapher-helper
  (root-name-list edge-list callback-function-click
   callback-function-click-shift)
  ;; example: root-name-list: '("n1") edge-list:
  ;;   '(("n1" "n2") ("n1" "n3"))
  (setf edges edge-list
        roots root-name-list
        node-callback-click callback-function-click
        node-callback-click-shift callback-function-click-shift)
  (capi:contain

   (make-instance
    'column-layout
    :title "Entity Browser"
    :description
    (list
     (make-instance 'capi:graph-pane
                    :min-height 330
                    :max-height 420
                    :roots roots
                    :layout-function 'graph-layout
                    :children-function #'info-panel-node-children-helper
                    :edge-pane-function
                    #'(lambda(self from to)
                        (declare (ignore self))
                        (let ((prop-name ""))
                          (dolist (edge edge-list)
                            (if (and
```

```
                                        (equal from (first edge))
                                        (equal to (second edge)))
                                       (if (and (> (length edge) 2) (third edge))
                                           (let ((last-index
                                                  (search
                                                   "/" (third edge)
                                                   :from-end t)))
                                             (if last-index
                                                 (setf prop-name
                                                  (subseq (third edge)
                                                  (1+ last-index)))
                                               (setf prop-name (third edge)))))))
                                  (make-instance
                                   'capi:labelled-arrow-pinboard-object
                                   :data (format nil "~A" prop-name))))
                            :node-pinboard-class 'text-node
                            :input-model `(((:button-1 :release)
                                            ,#'(lambda (pane x y)
                                                 (handle-mouse-click-on-pane
                                                   pane x y)))
                                          ((:button-1 :release :shift) ;; :press)
                                           ,#'(lambda (pane x y)
                                                (handle-mouse-click-shift-on-pane
                                                  pane x y))))
                            :node-pane-function 'make-text-node)
                (setf
                 output-pane
                 (make-instance 'capi:collector-pane
                                :min-height 130
                                :max-height 220
                                :title "Message collection pane"
                                :text "..."
                                :vertical-scroll t
                                :horizontal-scroll t))))
              :title
              "Info Pane Browser: mouse click for info, mouse click + shift for web br\
owser"

              :best-width 550 :best-height 450)))
        (make-info-panel-grapher-helper h-root-name-list
          h-edge-list h-callback-function-click
          h-callback-function-shift-click))))
```

# Writing the UI

Returning to the file **src/kgn/gui.lisp**, we need to implement callback functions for handling mouse clicks on the **info-pane-panel**, showing the options popup panel, and handling the callback when the user wants to delete the local SQLite query cache:

```
1  (defun test-callback-click (selected-node-name)
2    (ignore-errors
3      (format nil "* user clicked on node: ~A~%" selected-node-name)))
4
5  (defun test-callback-click-shift (selected-node-name)
6    (ignore-errors
7      (if (equal (subseq selected-node-name 0 5) "<http")
8          (trivial-open-browser:open-browser
9           (subseq selected-node-name 1
10            (- (length selected-node-name) 1))))
11     (format
12       nil
13       "* user shift-clicked on node: ~A - OPEN WEB BROWSER~%"
14       selected-node-name)))
15
16 (defun cache-callback (&rest x) (declare (ignore x))
17   (if *USE-CACHING*
18       (capi:display
19         (make-instance 'options-panel-interface))))
20
21 (defun website-callback (&rest x)
22   (declare (ignore x))
23   (trivial-open-browser:open-browser
24     "http://www.knowledgegraphnavigator.com/"))
```

In lines 8-10 I am using a third party package **trivial-open-browser:open-browser** to open the default browser on your laptop. URIs in KGN have angle bracket characters around the URI so here we remove these characters. I also use this same function in lines 21-24 to show the user a web site that I built for this example application.

Again from **gui.lisp**, the following listing shows how to define the CAPI user interface and I refer you to the CAPI documentation for details:

```
(capi:define-interface kgn-interface ()
  ()
  (:menus
   (action-menu
    "Actions"
    (
     ("Copy generated SPARQL to clipboard"
      :callback
      #'(lambda (&rest x) (declare (ignore x))
          (let ((messages (capi:editor-pane-text text-pane2)))
            (capi::set-clipboard text-pane2
              (format nil "---- Generated SPARQL and comments:~%~%~A~%~%" messages)
              nil))))
     ("Copy results to clipboard"
      :callback
      #'(lambda (&rest x) (declare (ignore x))
          (let ((results (capi:editor-pane-text text-pane3)))
            (capi::set-clipboard text-pane2
              (format nil "---- Results:~%~%~A~%" results) nil))))
     ("Copy generated SPARQL and results to clipboard"
      :callback
      #'(lambda (&rest x) (declare (ignore x))
          (let ((messages (capi:editor-pane-text text-pane2))
                (results (capi:editor-pane-text text-pane3)))
            (capi::set-clipboard
             text-pane2
             (format nil
               "---- Generated SPARQL and comments:~%~%~A~%~%---- Results:~%~%~A~%"
               messages results) nil))))
     ("Visit Knowledge Graph Navigator Web Site" :callback 'website-callback)
     ("Clear query cache" :callback 'cache-callback)
     ((if *show-info-pane*
          "Stop showing Grapher window for new results"
        "Start showing Grapher window for new results")
      :callback 'toggle-grapher-visibility)
     )))
  (:menu-bar action-menu)
  (:panes
   (text-pane1
    capi:text-input-pane
    :text (nth (random (length *examples*)) *examples*)
    :title "Query"
    :min-height 80
```

```
44      :max-height 100
45      :max-width *width*
46      ;;:min-width (- *width* 480)
47      :width *best-width*
48      :callback 'start-progress-bar-test-from-background-thread)
49
50     (progress-bar
51      capi:progress-bar
52      :start 0
53      :end 100
54      )
55
56     (text-pane2
57      capi:collector-pane
58      :font "Courier"
59      :min-height 210
60      :max-height 250
61      :title "Generated SPARQL queries to get results"
62      :text "Note: to answer queries, this app makes multipe SPARQL queries to DBPedia\
63  . These SPARQL queries will be shown here."
64      :vertical-scroll t
65      :create-callback #'(lambda (&rest x)
66                           (declare (ignore x))
67                           (setf (capi:editor-pane-text text-pane2) *pane2-message*))
68      :max-width *width*
69      :width *best-width*
70      :horizontal-scroll t)
71
72     (text-pane3
73      capi:collector-pane ;; capi:display-pane ;; capi:text-input-pane
74      :text *pane3-message*
75      :font "Courier"
76      :line-wrap-marker nil
77      :wrap-style :split-on-space
78      :vertical-scroll :with-bar
79      :title "Results"
80      :horizontal-scroll t
81      :min-height 220
82      :width *best-width*
83      :create-callback #'(lambda (&rest x)
84                           (declare (ignore x))
85                           (setf (capi:editor-pane-text text-pane3) *pane3-message*))
86      :max-height 240
```

```
87      :max-width *width*)
88     (info
89      capi:title-pane
90      :text "Use natural language queries to generate SPARQL"))
91    (:layouts
92     (main-layout
93      capi:grid-layout
94      '(nil info
95        nil text-pane1
96        nil text-pane2
97        nil text-pane3
98        nil progress-bar)
99       :x-ratios '(1 99)
100     :has-title-column-p t))
101   (:default-initargs
102    :layout 'main-layout
103    :title "Knowledge Graph Navigator"
104    :best-width *best-width*
105    :max-width *width*))
```

I showed you how to run the KGN example application earlier and I suggest that you leave the application open when reading through the user interface code.

For most of the development of KGN, the code layout and control flow was fairly simple. After the application was complete however, I noticed a bad user interface problem: making many calls to the DBPedia service took time and the application and except for streaming output to the generated SPARQL pane the application does nothing for a while which could confuse users. I decided to add a progress bar at the bottom of the main window and extracted much of the query processing functionality to a work thread, as implemented in the following listing, and pass a "update progress bar" callback function to many of the helper functions that create the SPARQL queries, make the web calls, and process the results. This callback function moves the progress bar. This complexity makes the KGN code is not as good a book example, but makes the application much better. The following function is derived from a multi-processing LispWorks example program. The local function **update-progress-bar** defined in the special operator **flet** in lines 4-8 is the function **updater** passed into functions we have seen earlier. This function updates the progress bar and is called during long running function calls. **flet** is like a **let** that additionally allows definitions of functions that inherit the local content of any variables defined in the **flet**.

```
(defun start-progress-bar-test-from-background-thread (query-text self)
  (with-slots (text-pane2 text-pane3  progress-bar) self
    (print text-pane2)
    (flet ((update-progress-bar (percent)
             (capi:execute-with-interface
              self
              #'(lambda ()
                  (setf (capi:range-slug-start progress-bar) percent)))))
      (mp:process-run-function "progress-bar-test-from-background-thread"
                               '()
                               'run-and-monitor-progress-background-thread
                               #'update-progress-bar
                               query-text text-pane2 text-pane3
                              ))))

(defvar *percent*)

(defun run-and-monitor-progress-background-thread
                       (updater text text-pane2 text-pane3)
  (setf *percent* 0)
  (unwind-protect
      (setf (capi:editor-pane-text text-pane2) "")
    (setf (capi:editor-pane-text text-pane3) "")
    ;;(capi:display-message "done")
    (let ((message-stream (collector-pane-stream text-pane2))
          (results-stream (collector-pane-stream text-pane3)))
      (format message-stream "# Starting to process query....~%")
      (format results-stream *pane3-message*)
      (let ((user-selections
             (get-entity-data-helper text
                :updater updater
                :message-stream message-stream)))
        (setf *percent* (+ *percent* 2))
        (funcall updater *percent*)
        (setf (capi:editor-pane-text text-pane3) "")
        (dolist (ev user-selections)
          (if (> (length (cadr ev)) 0)
              (let ()
                (terpri results-stream)
                (capi::write-string-with-properties
                 (format nil "- - - ENTITY TYPE: ~A - - -" (car ev))
                 '(:highlight :compiler-error-highlight) results-stream)
                (terpri results-stream)
```

```
                (dolist (uri (cadr ev))
                  (setf uri (car uri))
                  (case (car ev)
                    (:people
                     (pprint-results
                      (dbpedia-get-person-detail  uri :message-stream message-stream)
                      :stream results-stream))
                    (:companies
                     (pprint-results
                      (dbpedia-get-company-detail uri :message-stream message-stream)
                      :stream results-stream))
                    (:countries
                     (pprint-results
                      (dbpedia-get-country-detail uri :message-stream message-stream)
                      :stream results-stream))
                    (:cities
                     (pprint-results
                      (dbpedia-get-city-detail    uri :message-stream message-stream)
                      :stream results-stream))
                    (:products
                     (pprint-results
                      (dbpedia-get-product-detail uri :message-stream message-stream)
                      :stream results-stream))))))
          (setf *percent* (+ *percent* 1))
          (funcall updater *percent*))

        (let (links x)
          (dolist (ev user-selections)
            (dolist (uri (second ev))
              (setf uri (car uri))
              (if (> (length ev) 2)
                  (setf x (caddr ev)))
              (setf links (cons (list (symbol-name (first ev)) uri x) links))
              (setf *percent* (+ *percent* 1))
              (funcall updater *percent*)))

          (setf
            links
            (append
             links
             (entity-results->relationship-links
              user-selections
              :message-stream message-stream
```

```
87              :updater updater)))
88          (setf *percent* (+ *percent* 2))
89          (funcall updater *percent*)
90
91          (if
92            *show-info-pane*
93              (lw-grapher:make-info-panel-grapher
94                '("PEOPLE" "COMPANIES" "COUNTRIES" "CITIES"
95                  "PRODUCTS" "PLACES")
96                links 'test-callback-click
97                'test-callback-click-shift)))))
98    (funcall updater 0)))
```

We call the callback function **updater** at the end to remove the progress bar to let the user know that they can now enter another query.

If you have not already done so I hope you will take some time to download the LispWorks Personal Edition and try this application.

# Wrap-up

This is a long example application for a book so I did not discuss all of the code in the project. If you enjoy running and experimenting with this example and want to modify it for your own projects then I hope that I provided a sufficient road map for you to do so.

I got the idea for the KGN application because I was spending quite a bit of time manually setting up SPARQL queries for DBPedia (and other public sources like WikiData) and I wanted to experiment with partially automating this process. I wrote the CAPI user interface for fun since this example application could have had similar functionality as a command line tool. In fact, my first cut implementation was a command line tool with the user interface in the file **ui-text** that we looked at earlier. I decided to remove the command line interface and replace it using CAPI.

Most of the Common Lisp development I do has no user interface or implements a web application. When I do need to write an application with a user interface, the LispWorks CAPI library makes writing user interfaces fairly easy to do.

If you are using an open source Common Lisp like SBCL or CCL and you want to add a user interface then you might want to also try LTK[88] and McClim[89]. McClim works well on Linux and also works on macOS with XQuartz but with fuzzy fonts. I also like Radiance[90] that spawns a web browser so you can package web applications as desktop applications.

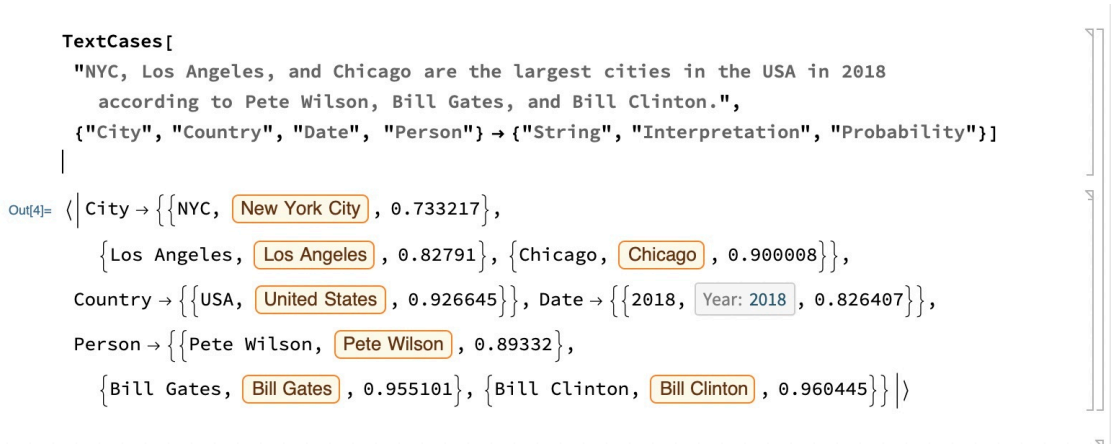
[88] http://www.peter-herth.de/ltk/
[89] https://www.cliki.net/McCLIM
[90] https://github.com/Shirakumo/radiance

If you are using CCL (Clojure Common Lisp) on macOS you can try the supported **COCOA-APPLICATION** package. This is only recommended if you already know the Cocoa APIs, otherwise this route has a very steep learning curve.

# Using Common Lisp With Wolfram/One

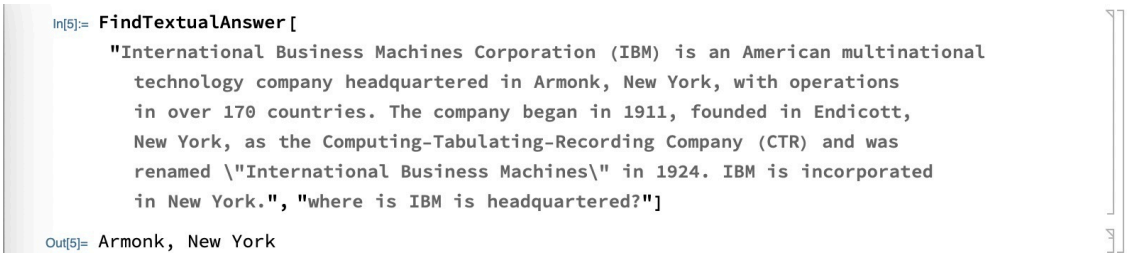
If you use Wolfram/One[91] then the material in this short chapter may interest you. The interface that I wrote is simple: I use **uiop:run-program** to spawn a new process to run the Wolfram Language command line tool that writes results to a temporary file. I then use **uiop:read-file-string** to read the results and parse them into a convenient form for use.

Before we build and use an interface to Wolfram/One, let's look at two screen shots of the Wolfram/One interface with examples that we will later run in Common Lisp. The first example finds entities in text:




**Using Wolfram/One to find entities in text**

The second example uses a deep learning model to answer a question given text containing the answer to the question:




**Using Wolfram/One to answer natural language questions**

Here is the **package.lisp** file for this example:

[91] https://www.wolfram.com/wolfram-one/

```
(defpackage #:wolfram
  (:use #:cl #:uiop)
  (:export #:wolfram #:cleanup-lists
           #:find-answer-in-text #:entities))
```

And the **wolfram.asd** file:

```
(asdf:defsystem #:wolfram
  :description "Wolfram Language interface experiments"
  :author "Mark Watson <markw@markwatson.com>"
  :license "Apache 2"
  :depends-on (#:uiop #:cl-json #:myutils)
  :components ((:file "package")
               (:file "wolfram")))
```

The implementation in **Wolfram.lisp** is simple enough. In lines 6-8 I create a Common Lisp *path* object in **/tmp** (and absolute pathname is required) and then use **file-namestring** to get just the file name as a string. In lines 8-10 we are creating an operating system shell and running the Wolfram Language command line tool with arguments to execute the query and write the results to the temporary file. In lines 11-15 we read the contents of the temporary file, delete the file, and decode the returned string as JSON data.

The Data returned form calling the Wolfram Language command line tool contains excess structure that we don't need (a sample of the raw returned data is shown later) so the function **cleanup-lists** shown in lines 17-19 discards heads of lists when the first value in a list or sublist is *Rule* or *List*. The function **recursive-remove** seen in lines 20-24 will remove all occurrences of an item from a nested list.

```
(in-package #:wolfram)

;; General query utilities

(defun wolfram (statement)
  (let ((temp-file-path
         (file-namestring (uiop:tmpize-pathname "/tmp/wolfram"))))
    (uiop:run-program (concatenate 'string "wolframscript -code 'Export[\""
                                   temp-file-path "\"," statement
                                   ",\"ExpressionJSON\"]'"))
    (let* ((ret (uiop:read-file-string temp-file-path)))
      (delete-file temp-file-path)
      (with-input-from-string (s (myutils:replace-all
                                  (myutils:replace-all ret "\"'" "\"") "'\"" "\""))
        (json:decode-json s)))))
```

```
16
17 (defun cleanup-lists (r)
18   (cdr (recursive-remove "Rule" (recursive-remove "List" r))))
19
20 (defun recursive-remove (item tree)
21   (if (atom tree)
22       tree
23       (mapcar (lambda (nested-list) (recursive-remove item nested-list))
24               (remove item tree :test #'equal))))
25
26 ;; Higher level utilities for specific types of queries
27
28 (defun entities (text)
29   (let* ((noquotes (myutils:replace-all (myutils:replace-all text "\"" " ") "'" " "))
30          (query2
31           (concatenate
32            'string "TextCases['" noquotes
33            "',  {'City', 'Country', 'Date', 'Person'} ->"
34            " {'String', 'Interpretation', 'Probability'}]"))
35          (query (myutils:replace-all query2 "'" "\"")))
36     (remove-if #'(lambda (a) (null (cadr a)))
37                (cleanup-lists (wolfram query)))))
38
39 (defun find-answer-in-text (text question)
40   (let* ((nqtext (myutils:replace-all (myutils:replace-all text "\"" " ") "'" " "))
41          (nqquestion (myutils:replace-all
42                       (myutils:replace-all question "\"" " ") "'" " "))
43          (query2 (concatenate 'string "FindTextualAnswer['" nqtext
44                               "', '" nqquestion "']"))
45          (query (myutils:replace-all query2 "'" "\"")))
46     (wolfram query)))
```

The last two functions in the last code listing, **entities** and **find-answer-in-text** are higher level functions intended to work with the Wolfram Language procedures **TextCases** (see Wolfram documentation for TextCases[92]) and **FindTextualAnswer** (see Wolfram documentation for FindTextualAnswer[93]).

The functions **cleanup-lists** and **recursive-remove** can be used to clean up results. First, we will just call function **wolfram** and show the raw results:

[92] https://reference.wolfram.com/language/ref/TextCases.html
[93] https://reference.wolfram.com/language/ref/FindTextualAnswer.html

```
$ sbcl
* (ql:quickload "wolfram")
To load "wolfram":
  Load 1 ASDF system:
    wolfram
; Loading "wolfram"
[package myutils]................................
[package wolfram]
("wolfram")
* (setf example "TextCases['NYC, Los Angeles, and Chicago are the largest cities in \
the USA in 2018 according to Pete Wilson.', {'City', 'Country', 'Date', 'Person'} ->\
 {'String', 'Interpretation', 'Probability'}]")
"TextCases['NYC, Los Angeles, and Chicago are the largest cities in the USA in 2018 \
according to Pete Wilson.', {'City', 'Country', 'Date', 'Person'} -> {'String', 'Int\
erpretation', 'Probability'}]"
* (setf example-str (myutils:replace-all  example "'" "\""))
"TextCases[\"NYC, Los Angeles, and Chicago are the largest cities in the USA in 2018\
 according to Pete Wilson.\", {\"City\", \"Country\", \"Date\", \"Person\"} -> {\"St\
ring\", \"Interpretation\", \"Probability\"}]"
* (setf results (wolfram:wolfram example-str))
* (pprint results)

("Association"
 ("Rule" "City"
  ("List"
   ("List" "NYC" ("Entity" "City" ("List" "NewYork" "NewYork" "UnitedStates"))
    0.75583166)
   ("List" "Los Angeles"
    ("Entity" "City" ("List" "LosAngeles" "California" "UnitedStates"))
    0.84206486)
   ("List" "Chicago"
    ("Entity" "City" ("List" "Chicago" "Illinois" "UnitedStates"))
    0.91092855)))
 ("Rule" "Country"
  ("List" ("List" "USA" ("Entity" "Country" "UnitedStates") 0.9285077)))
 ("Rule" "Date"
  ("List"
   ("List" "2018" ("DateObject" ("List" 2018) "Year" "Gregorian" -7.0)
    0.8364356)))
 ("Rule" "Person"
  ("List"
   ("List" "Pete Wilson" ("Entity" "Person" "PeteWilson::s7259") 0.9274548))))
*
```

Now we clean up the output:

```
* (defvar results-cleaned (wolfram:cleanup-lists results))
* (pprint results-cleaned)

(("City"
  (("NYC" ("Entity" "City" ("NewYork" "NewYork" "UnitedStates")) 0.75583166)
   ("Los Angeles" ("Entity" "City" ("LosAngeles" "California" "UnitedStates"))
    0.84206486)
   ("Chicago" ("Entity" "City" ("Chicago" "Illinois" "UnitedStates"))
    0.91092855)))
 ("Country" (("USA" ("Entity" "Country" "UnitedStates") 0.9285077)))
 ("Date" (("2018" ("DateObject" (2018) "Year" "Gregorian" -7.0) 0.8364356)))
 ("Person" (("Pete Wilson" ("Entity" "Person" "PeteWilson::s7259") 0.9274548))))
*
```

Next we will try the two higher-level utility functions. The first example shows finding entities in text:

```
CL-USER 21 > (pprint
               (wolfram:entities "Sedona Arizona is home to Mark Louis Watson"))

(("City"
  (("Sedona" ("Entity" "City" ("Sedona" "Arizona" "UnitedStates")) 0.8392784)))
 ("Person" (("Mark Louis Watson" "Mark Louis Watson" 0.9023427))))
```

The second example uses a Wolfram pre-trained deep learning model for question answering:

```
CL-USER 22 > (pprint
               (wolfram::find-answer-in-text "International Business Machines Corpor\
ation (IBM) is an American multinational technology company headquartered in Armonk,\
 New York, with operations in over 170 countries. The company began in 1911, founded\
 in Endicott, New York, as the Computing-Tabulating-Recording Company (CTR) and was \
renamed \"International Business Machines\" in 1924. IBM is incorporated in New York\
."
                           "where is IBM is headquartered?"))

"Armonk, New York"
```

If you use Wolfram/One then these examples should get you started wrapping other Wolfram Language functionality for use in your Common Lisp applications.

# Book Wrapup

Congratulations for finishing this book!

I love programming in Lisp languages with concise code and a bottom-up approach to development. I hope you now also share this enthusiasm with me.

Common Lisp is sometimes criticised as not having as many useful libraries as some newer languages like Python and Java, and this is a valid criticism. That said, I hope the wide variety of examples in this book will convince you that Common Lisp is a good choice for many types of programming projects.

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 fourth 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.

Thank you for buying and reading my book!

Mark Watson