

Practical Semantic Web and Linked Data Applications

Common Lisp Edition

Mark Watson

Copyright 2010 Mark Watson. All rights reserved.

This work is licensed under a Creative Commons
Attribution-Noncommercial-No Derivative Works

Version 3.0 United States License.

**PRE-RELEASE VERSION FOR BETA TEST READERS:
NOT FOR RELEASE**

February 26, 2010

Contents

Preface	ix
1 AllegroGraph Embedded Lisp Quick Start	1
1.1 Starting AllegroGraph	1
1.2 Working with RDF Data Stores	2
1.2.1 Creating Repositories	3
1.2.2 AllegroGraph Lisp Reader Support for RDF	4
1.2.3 Adding Triples	4
1.2.4 Fetching Triples by ID	5
1.2.5 Printing Triples	5
1.2.6 Using Cursors to Iterate Through Query Results	7
1.2.7 Saving Triple Stores to Disk as XML, N-Triples, and N3	7
1.3 AllegroGraph's Extensions to RDF	8
1.3.1 Examples Using Triple and Graph IDs	8
1.3.2 Support for Geo Location	10
1.3.3 Support for Free Text Indexing	12
1.3.4 Comparing AllegroGraph With Other Semantic Web Frameworks	13
1.4 AllegroGraph Quickstart Wrap Up	14
I Semantic Web Technologies	15
2 Linked Data and the World Wide Web	17
2.1 Linked Data Resources on the Web	17
2.2 Graph Visualization of Linked Data using Gruff	17
2.3 The Future: RDFa?	18
3 RDF	19
3.1 RDF Examples in N-Triple and N3 Formats	20
3.2 The RDF Namespace	23
3.2.1 rdf:type	23
3.2.2 rdf:Property	23
3.3 RDF Wrap Up	24
4 RDFS	25
4.1 Extending RDF with RDF Schema	25

4.2	Modeling with RDFS	26
4.3	TBox and ABox	28
4.4	Descriptive Logic Classes are Derivable	28
4.5	AllegroGraph RDFS++ Extentions	28
4.6	RDFS Wrapup	28
5	The SPARQL Query Language	29
5.1	Example RDF Data in N3 Format	29
5.2	Example SPARQL SELECT Queries	32
5.3	Example SPARQL CONSTRUCT Queries	34
5.4	Example SPARQL ASK Queries	34
5.5	Example SPARQL DESCRIBE Queries	34
6	RDFS++ and OWL	35
6.1	Properties Supported In RDFS++	35
6.1.1	owl:sameAs	36
6.1.2	owl:inverseOf	36
6.1.3	owl:TransitiveProperty	36
6.2	RDF, RDFS, and RDFS++ Modeling Wrap Up	37
II	AllegroGraph Extended Tutorial	39
7	SPARQL Queries Using AllegroGraph APIs	41
7.1	Using Namespaces	41
7.2	Reading RDF Data From Files	42
7.3	Lisp APIs for Queires	42
7.4	Wrap Up	44
8	AllegroGraph Reasoning System	45
8.1	Enabling RDFS++ Reasoning on a Triple Store	45
8.2	Inferring New Triples: rdf:type vs. rdfs:subClassOf Example	46
8.3	Controlling AllegroGraph’s Inferencing Engine	47
9	AllegroGraph Prolog Interface	49
III	Common Lisp Utilities for Information Processing	51
10	Common Lisp Client Library for Open Calais	53
10.1	Open Calais Web Services Client	53
10.2	Storing Entity Data in an RDF Data Store	55
10.3	Testing the Open Calais Demo System	57
10.4	Open Calais Wrap Up	58

11 Common Lisp Client Library for Natural Language Processing	59
11.1 KnowledgeBooks.com Natural Language Processing Library	59
11.2 Entity Extraction from Text	59
11.2.1 Entity Extraction from Text Library	59
11.2.2 Entity Extraction with AllegroGraph Example	60
11.3 Automatic Text Tagging	60
11.3.1 KnowledgeBooks.com Text Tagging Library	60
11.3.2 Text Tagging with AllegroGraph Example	60
11.4 Automatically Summarizing Text	60
11.4.1 KnowledgeBooks.com Automatic Summarization Library	60
11.4.2 Automatic Summarization and AllegroGraph Index/Search Example	60
12 Common Lisp Client Library for Freebase	61
12.1 Overview of Freebase	61
12.1.1 MQL Query Language	61
12.1.2 Search	61
12.1.3 Geo Search	61
12.2 Freebase Common Lisp Client Library	62
13 Common Lisp Client Library for DBpedia	63
13.1 Using the AllegroGraph SPARQL Client Library to access DBpedia	63
13.2 Example SPARQL Queries Using DBpedia	63
14 Common Lisp Client Library for Exporting SPARQL Query Results	65
14.1 Exporting to OpenOffice.org Spreadsheets	65
14.2 Exporting to PostgreSQL Databases	65
IV Sample Application: KnowledgeBooks Semantic Web Spider	67
15 Semantic Web Spider: Information Gathering	69
16 Semantic Web Spider: RDF Data Storage	71
17 Semantic Web Spider: APIs	73
V Sample Application: Semantic Web Enabled Web Site	75
18 Requirements and Design of the Semantic Web Portal Web Application	77
19 Web Application Back End Implementation	79

- 20 Web Interface for the Semantic Web Portal** **81**
- 20.1 Introduction to AllegroServe 81
- 20.2 Introduction to Web Actions 81
- 20.3 Dojo and Javascript for the Web Interface 81
- 20.4 Web Application Implementation 81

- Bibliography** **83**

List of Figures

List of Figures

List of Tables

List of Tables

Preface

This book is intended to be a practical guide for using RDF data in information processing, linked data, and semantic web applications using the Common Lisp APIs for the AllegroGraph product.

This book has a companion edition that covers the use of both AllegroGraph and the open source Sesame project using JVM based languages like Java, Clojure, JRuby, and Scala.

I take an unusual approach in both Lisp and Java editions of this book: instead of digging too deeply into proprietary APIs for available data stores (for example, AllegroGraph, Jena, Sesame, 4Store, etc.) we will concentrate on a more standards-based approach: we will deal with RDF data stored in easy to read N-Triple and N3 formats and perform all queries using the standard SPARQL query language. I am much more interested in showing you how to model data with RDF and write practical applications.

AllegroGraph is written in Common Lisp and comes in several "flavors":

1. As a standalone server that supports Lisp, Ruby, Java, Clojure, Scala, and Python clients. A free version (limited to 50 million RDF triples - a large limit) that can be used for any purpose, including commercial use. This book (Volume I) primarily deals with the server version of AllegroGraph.
2. The WebView interface for exploring, querying, and managing AllegroGraph triple stores. WebView is standalone because it contains an embedded AllegroGraph server. TBD: describe licensing of free version. Covered in Volume II.
3. The Gruff for exploring, querying, and managing AllegroGraph triple stores using table and graph views. Gruff is standalone because it contains an embedded AllegroGraph server. TBD: describe licensing of free version. Covered in Volume II.
4. A library that is used embedded in Franz Common Lisp applications. A free version is available (with some limitations) for non-commercial use. Covered in Volume II.

Regardless of which programming languages that you use, the basic techniques of using AllegroGraph are very similar.

Preface

The small example code snippets in the book text are in the public domain. The larger example applications and libraries are licensed using the AGPL. As an individual developer, if you purchase the print edition of this book then I give you a commercial use waiver to the AGPL deploying your applications: you can use my examples in commercial applications without the requirement of releasing the source code for your application under the AGPL. If you work for a company that would like use my examples with a commercial use waiver, then have your company purchase two print copies of this book for use by your development team.

Acknowledgements

People who contributed technical ideas for this book

Gary King (Franz)

People who contributed technical edits for this book

TBD

People who contributed copy edits to the material for this book

Carol Watson

1 AllegroGraph Embedded Lisp Quick Start

The first section of this book will cover Semantic Web technologies from a theoretical and reference point of view. While covering the theory it will be useful to provide some concrete examples using AllegroGraph so this book is organized in layers:

1. Quick introduction to the AllegroGraph and Franz Lisp.
2. Theory (with some AllegroGraph-specific short examples).
3. Detailed treatment of AllegroGraph APIs.
4. Development of Useful Common Lisp libraries information processing, and importing linked data sources like Freebase and Open Calais data to an AllegroGraph RDF store.
5. Development of a complete web portal using Semantic Web technologies.

It will be easier to work through the theory in Chapters 3, 4, and 6 if you understand the basics of AllegroGraph. After a detailed look theory we will dig deeper into AllegroGraph development techniques in Chapters 7, 8, and 9.

1.1 Starting AllegroGraph

In this chapter and in much of this book, you can save some effort by copy and pasting the code snippets into the Lisp listener. The code snippets used in this chapter are contained in the source file **quick.start.lisp.embedded.lisp**. I am going to assume that most readers are trying AllegroGraph using the free non-commercial use version so that is what I will use here. If you are using a commercially licensed version the examples will work the same but the initial banner display by *alisp* (conventional case insensitive Lisp shell) and *mlisp* (“modern” case sensitive Lisp shell) will be slightly different. While I usually use *alisp* in my work (I have been using Lisp for professional development since 1982), Franz recommends using *mlisp* for AllegroGraph development so we will use *mlisp* in this book. You will need to following the directions in `acl81.express/readme.txt` to build a *mlisp* image to use. When showing interactive examples in this chapter I remove some Lisp shell messages so when you

work along with these examples expect to see more output than what is shown here¹:

```
markw$ mlisp
International Allegro CL Free Express Edition
8.1 [Mac OS X (Intel)] (Jul 9, 2009 17:15)
Copyright (C) 1985-2007, Franz Inc., Oakland, CA, USA.
    All Rights Reserved.
```

This development copy of Allegro CL is licensed to:
Trial User

```
;; Current reader case mode: :case-sensitive-lower
cl-user(1): (require :agraph)
AllegroGraph Lisp Edition 3.2 [built on March 16, 2009 15:05:15
t
cl-user(2): (in-package :db.agraph.user)
#<The db.agraph.user package>
TRIPLE-STORE-USER(3):
```

Here I required the **:agraph** package and changed the current Common Lisp package to **db.agraph.user**. In examples later in this book when we develop complete application examples we will be using our own application-specific packages and I will show you then what you need in general to import from *db.agraph* and *db.agraph.user*. We will continue this interactive example Lisp session in the following sections.

While I show interactive sessions in the examples in this book, if you are a Windows user then you will may want to use the Windows-specific IDE. I recommend that OS X, Linux, and Windows users use Emacs to develop Lisp code².

1.2 Working with RDF Data Stores

RDF data stores provide the services for storing RDF triple data and providing some means of making queries to identify some subset of the triples in the store. I think that it is important to keep in mind that the mechanism for maintaining triple stores varies in different implementations. Triples can be stored in memory, in disk-based btree stores like BerkeleyDB, in relational databases, and in custom stores like AllegroGraph. While much of this book is specific to Common Lisp and AllegroGraph

¹I use OS X and Linux for my development. If you are a Windows user, follow the installation instructions on the AllegroGraph download web page and expect to see slight differences to the interactive example sessions that I use in this book.

²Franz provides their own Emacs tools: look for instructions for installing ELI. However, I use the SLIME Emacs Lisp development tools that are compatible with all versions of Lisp that I use: Franz, SBCL, ClozureCL, and Gambit-C Scheme. Franz provides SLIME installation instructions for Franz Common Lisp

the concepts that you will learn and experiment with can be useful if you also use other languages and platforms like Java (Sesame, Jena, OwlAPIs, etc.), Ruby (Redland RDF), etc. For Java developers Franz offers a Java version of AllegroGraph (implemented in Lisp with a network interface that also supports Python and Ruby clients) that I will cover in Volume I of this book.

1.2.1 Creating Repositories

AllegroGraph uses disk-based RDF storage with automatic in-memory caching. For the examples in this book I will assume that all RDF stores are kept in the temporary directory `/tmp`. For deployed systems you will clearly want to use a permanent location. For Windows(tm) development you can either change this location or create a new directory in `c:\tmp`. In the examples in this book, I assume a Mac OS X, Linux, or other Unix type file system:

```
TRIPLE-STORE-USER(3): (create-triple-store
                        "/tmp/rdfstore_1")
#<db.agraph::triple-db /tmp/rdfstore_1, open @ #x109682>
```

While it is possible to simultaneously work with multiple repositories (and this is well documented in Franz's online documentation for the non-free versions of AllegroGraph) for all of the tutorials, examples, and sample applications in this book we need just a single open repository in order to be compatible with the free versions of AllegroGraph.

We will see in Chapter 3 how to partition RDF triples into different namespaces and to use existing RDF data and schemas in different namespaces. For now, I introduce the AllegroGraph APIs for defining new namespaces and listing all namespaces defined in the current repository:

```
TRIPLE-STORE-USER(4): (register-namespace "kb"
                      "http://knowledgebooks.com/rdfs#")
"http://knowledgebooks.com/rdfs#"
TRIPLE-STORE-USER(5): (display-namespaces)
rdfs => http://www.w3.org/2000/01/rdf-schema#
err => http://www.w3.org/2005/xqt-errors#
fn => http://www.w3.org/2005/xpath-functions#
rdf => http://www.w3.org/1999/02/22-rdf-syntax-ns#
xs => http://www.w3.org/2001/XMLSchema#
xsd => http://www.w3.org/2001/XMLSchema#
owl => http://www.w3.org/2002/07/owl#
kb => http://knowledgebooks.com/rdfs#
```

Here I created a new name space that has an abbreviation (or nickname) **kb:** and then printed out all registered namespaces. To insure data integrity be sure to call (**close-triple-store**) to close an RDF triple store when you are done with it.

1.2.2 AllegroGraph Lisp Reader Support for RDF

In general, the subject, predicate, and object parts of an RDF triple can be either URIs or literals.

AllegroGraph provides a Lisp reader macro **!** that makes it easier to enter URIs and literals. For example, the following two URIs are functionally equivalent given the (**register-namespace “kb” ...**) in the last section:

```
!<http://knowledgebooks.com/rdfs#containsPerson>  
!kb:containsPerson
```

String literals are also defined using the **!** reader macro; for example:

```
!"Barack Obama"  
!"101 Main Street"
```

1.2.3 Adding Triples

A triple consists of a subject, predicate, and object. We refer to these three values as symbols **:s**, **:p**, and **:o** when using the AllegroGraph APIs. We saw the use of literals with the **!** Lisp reader macro in the last section. If we need to refer to either a subject, predicate, or object as a web URI then we use the function **resource**:

```
TRIPLE-STORE-USER(15): (resource "http://demo_news/12931")  
!<http://demo_news/12931>  
TRIPLE-STORE-USER(16): (defvar *demo-article*  
                          (resource  
                           "http://demo_news/12931"))  
*demo-article*  
TRIPLE-STORE-USER(17): *demo-article*  
!<http://demo_news/12931>
```

The function **add-triple** takes three arguments for the subject, predicate, and object in a triple:

```
TRIPLE-STORE-USER(18): (add-triple *demo-article*
                        !rdf:type
                        !kb:article)
1
TRIPLE-STORE-USER(19): (add-triple *demo-article*
                        !kb:containsPerson
                        !"Barack Obama")
2
```

We used a combination of a generated resource, two predicates defined in the **rdf:** and **kb:** namespaces, and a string literal to define two triples. You notice that the function **add-triple** returns an integer as its value: this is a unique ID for the newly created triple.

1.2.4 Fetching Triples by ID

Triples in an AllegroGraph RDF store can be identified by a unique ID; this ID value is returned as the value of calling **add-triple** and can be used to fetch a triple:

```
TRIPLE-STORE-USER(20): (get-triple-by-id 2)
<12931 containsPerson Barack Obama>
TRIPLE-STORE-USER(21): (defvar *triple*
                        (get-triple-by-id 2))
*triple*
TRIPLE-STORE-USER(22): *triple*
<12931 containsPerson Barack Obama>
```

We will seldom access triples by ID – shortly we will see how to query a RDF store to find triples.

1.2.5 Printing Triples

The function **print-triple** can be used to print a short form of a triple and also by adding the arguments **:format** **:concise** we can print a triple in the NTriple format:

```
TRIPLE-STORE-USER(23): (print-triple *triple*
                                :format :concise)
<4: http://demo_news/12931 kb:containsPerson
    Barack Obama>
<12931 containsPerson Barack Obama>
```

```
TRIPLE-STORE-USER(24): (print-triple *triple*)
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#containsPerson>
  "Barack Obama" .
<12931 containsPerson Barack Obama>
```

Function **print-triple** prints a triple to standard output and returns the triple value in the short notation. We will see in later chapters how to create something like a database cursor for iterating through multiple triples that we find by querying a triple store. For now we will use query function **get-triples** that returns all triples matching a query in a list. The utility function **print-triples** prints all triples in a list:

```
TRIPLE-STORE-USER(27): (print-triples (list *triple*))
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#containsPerson>
  "Barack Obama" .
TRIPLE-STORE-USER(28): (print-triples (get-triples))
<http://demo_news/12931>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://knowledgebooks.com/rdfs#article> .
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#containsPerson>
  "Barack Obama" .
```

When **get-triples** is called with no arguments it simple returns all triples in a data store. We can specify query matching values for any combination of `:s`, `:p`, and `:o`. We can look at all triples that have their subject equal to the resource we created for the demo article:

```
TRIPLE-STORE-USER(31): (print-triples
                        (get-triples :s *demo-article*))
<http://demo_news/12931>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://knowledgebooks.com/rdfs#article> .
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#containsPerson>
  "Barack Obama" .
```

We can limit query results further; in this case we add the condition that the object must equal the value of the type **!kb:article**:

```
TRIPLE-STORE-USER(33): (print-triples
```

```
(get-triples :s *demo-article*
             :o !kb:article))
<http://demo_news/12931>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://knowledgebooks.com/rdfs#article> .
```

I often need to manually reformat program example text and example program output in this book. The last three lines in the last example would appear on a single line if you are following along with these tutorial examples in a Lisp listener (as you should be!) In any case RDF triple data in the NTriple format that we are using here is free-format: a triple is defined by three tokens (each with no embedded whitespace unless inside a string literal) and ended with a period character.

1.2.6 Using Cursors to Iterate Through Query Results

```
TRIPLE-STORE-USER(39): (setq a-cursor (get-triples
                                     :s *demo-article*))
#<DB.AGRAPH::FILTERED-CURSOR
 #<DB.AGRAPH::ROW-CURSOR
  #<DB.AGRAPH::TRIPLE-RECORD-FILE @ #x113fd61a> ...
  #x11672082>
 @ #x1167219a>
TRIPLE-STORE-USER(40): (while (cursor-next-p a-cursor)
 ; cursor-next returns a vector, not a triple:
 (print (cursor-next-row a-cursor)))

<12931 type article>
<12931 containsPerson Barack Obama>
NIL
TRIPLE-STORE-USER(41):
```

1.2.7 Saving Triple Stores to Disk as XML, N-Triples, and N3

It is often useful to copy either all triples in data store or triples matching a query out to a flat disk file in N-Triples format:

```
(with-open-file (output "/tmp/sample.ntriple"
                    :direction :output
                    :if-does-not-exist :create)
  (print-triples (get-triples)
                 :stream output :format :ntriple))
```

Output in the file might look like:

```
<http://demo_news/12931>
  <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
  <http://knowledgebooks.com/rdfs#article> .
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#containsPerson>
  "Barack Obama" .
```

Here we see two triples in N-Triple format. In most applications the RDF data store will be persistent and reused over multiple application restarts. While the disk based triple store is persistent for many applications it is a good idea to support exporting triples in a standard format like the N-Triple format that we use here, the XML serialization format, or the newer and more compact N3 format.

1.3 AllegroGraph's Extensions to RDF

We have seen that RDF triples contain three values: *subject*, *predicate*, and *object*. We will cover this more in Chapter 3. AllegroGraph extends RDF adding two additional values:

1. graph-id – optional string to specify which graph the RDF triple belongs to
2. triple-id – unique triple ID

The *subject*, *predicate*, *object*, and *graph* value strings are uniquely stored in a global string table (like the symbol table a compiler uses) so that triples can more efficiently store indices rather than complete strings. Storing just a single copy of each unique string also save memory and disk storage. Comparing string table indices is also much faster than storing string values.

1.3.1 Examples Using Triple and Graph IDs

In the following example we will extend the example started in Chapter 1 by adding an additional triple specifying an optional graph ID value and the value for the RDF data store:

```
(require :agraph)
(in-package :db.agraph.user)

(create-triple-store "/tmp/rdfstore_1")
```

```
;; default data store is kept in *db*
*db*
```

The value of ***db*** prints as:

```
#<DB.AGRAPH::TRIPLE-DB /tmp/rdfstore_1,
      open @ #x11790d02>
```

After registering a namespace we add three triples. Unlike the examples seen earlier in this chapter, we specify values for two optional parameters to function **add-triple**:

```
(register-namespace "kb"
                   "http://knowledgebooks.com/rdfs#")
(resource "http://demo_news/12931")
(defvar *demo-article*
  (resource "http://demo_news/12931"))

(add-triple *demo-article* !rdf:type !kb:article
            :db *db* :g !"news-data")
(add-triple *demo-article* !kb:containsPerson !"Barack Obama"
            :db *db* :g !"news-data")
(add-triple *demo-article* !kb:processed !"yes"
            :db *db* :g !"work-flow")
```

In addition to queries based on values of subject, predicate, and object we can also filter results by specifying a value for the graph:

```
;; query on optional graph value:
(print-triples (get-triples :g !"work-flow"))
```

producing the output:

```
<http://demo_news/12931>
  <http://knowledgebooks.com/rdfs#processed>
  "yes" .
```

For the last three triples that we added to the triple store we used the optional **:db** keyword argument for function **add-triple**. Because we used the triple store stored in the global variable ***db*** using this optional keyword argument had no effect. However, it is possible to have multiple triple stores open at the same time so it can make sense to partition RDF data over multiple data stores on different servers. We will not

concern ourselves in this book (except for mentioning it here) with AllegroGraph's client API functionality to access multiple distributed AllegroGraph servers. Franz's online documentation covers how to use the federation mechanism.

The function **add-triple** returns as its value the newly created triple's ID and has the side effect of adding the triple to the currently opened data store. While it is not best practice to use this unique internal AllegroGraph triple ID as a value referenced in another triple, there may be reasons in an application to store the IDs of newly created triples in order to be able to retrieve them from ID; for example:

```
TRIPLE-STORE-USER(15) : (get-triple-by-id 3)
<12931 processed yes work-flow>
```

1.3.2 Support for Geo Location

Geo Location support in AllegroGraph is more general than 2D map coordinates or other 2D coordinate systems. Since I will not be using Geo Location in later examples in this book, I will briefly introduce you to the Geo Location APIs and then refer you to Franz's online documentation. The example code snippets for this section are found in the file `quick_start.allegrograph.lisp.embedded/geoloc.lisp`:

```
(require :agraph)

(in-package :db.agraph.user)
(enable-!-reader)
(register-namespace "g" "http://knowledgebooks.com/geo#")
(create-triple-store "/tmp/geospatial-test")

;; define some locations in Verde Valley, Arizona:
(defvar *locs*
  '(("Verde_Valley_Ranger_Station" 34.7666667 -112.1416667)
    ("Verde_Valley_School" 34.8047596 -111.8060388)
    ("Sedona" 34.8697222 -111.7602778)
    ("Cottonwood" 34.739 -112.009)
    ("Jerome" 34.75 -112.11)
    ("Flagstaff" 35.20 -111.63)
    ("Clarkdale" 34.76 -112.05)
    ("Mount_Wilson" 35.996 -114.611)
    ("Tuzigoot" 34.56 -111.84)))

(defvar *min-lat* (reduce #'min (mapcar #'cadr *locs*)))
(defvar *max-lat* (reduce #'max (mapcar #'cadr *locs*)))
(defvar *min-lon* (reduce #'min (mapcar #'caddr *locs*)))
```

```

(defvar *max-lon* (reduce #'max (mapcar #'caddr *locs*)))

;; create a type:
(setf offset 5.0)
(flet ((fixup (num direction)
        (if (eq direction :min)
            (- num offset)
            (+ num offset))))
  (setf *verde-valley-arizona*
    (db.agraph:register-latitude-striping-in-miles
      3
      :lat-min (fixup *min-lat* :min)
      :lat-max (fixup *max-lat* :max)
      :lon-min (fixup *min-lon* :min)
      :lon-max (fixup *max-lon* :max))))

(add-geospatial-subtype-to-db *verde-valley-arizona*)

(dolist (loc *locs*)
  (let ((name (intern-resource
              (format nil "http://knowledgebooks.com/geo#~a"
                (car loc)))))
    (print name)
    (add-triple name !g:isAt3
      (longitude-latitude->upi *verde-valley-arizona*
        (caddr loc) (cadr loc)))))

(index-all-triples :wait t)

(print
  (count-cursor
    (get-triples-haversine-miles *verde-valley-arizona*
      !g:isAt3
      -112.009 34.739 ; longitude and latitude
      30.0) ; distance in miles

    (dolist (distance '(50.0 30.0 10.0 5.0))
      (format t "~%~%Checking with distance = ~A~%" distance)
      (let ((cursor
              (get-triples-haversine-miles
                *verde-valley-arizona* !g:isAt3
                -112.009 34.739 distance)))
        (while (cursor-next-p cursor)
          (print (cursor-next-row cursor)))))))

```

In this example, I print out the number locations in the triple store within 30 miles of the location (-112.009 34.739) and a list of all locations within 50, 30, 10, and 5 miles of this same location. Here is the part of the output for distances of 10 and 5 miles:

```
Checking with distance = 10.0
```

```
<Verde_Valley_Ranger_Station isAt3
                                     +344559.99894-1120830.01273>
<Jerome isAt3 +344500-1120636.00212>
<Clarkdale isAt3 +344535.99394-1120300.01091>
<Cottonwood isAt3 +344420.39424-1120032.40939>
```

```
Checking with distance = 5.0
```

```
<Clarkdale isAt3 +344535.99394-1120300.01091>
<Cottonwood isAt3 +344420.39424-1120032.40939>
```

1.3.3 Support for Free Text Indexing

The AllegroGraph support for free text indexing is very useful and we will use it often in this book. The example code snippets for this section are found in the file `quick_start.allegrograph.lisp.embedded/text.lisp`:

TBD: intersperse comments with code:

```
(require :agraph)
(in-package :db.agraph.user)

(enable-!-reader) ; enable the ! reader macro

(create-triple-store "/tmp/index_test")
(register-namespace "kb"
                    "http://knowledgebooks.com/rdfs#")

(register-freetext-predicate !kb:containsPerson)

;; print out all predicates registered for indexing subject and
;; in triples that use these predicates:
(freetext-registered-predicates)

(resource "http://demo_news/12931")
(defvar *demo-article*
```

```

(resource "http://demo_news/12931"))

(add-triple *demo-article* !rdf:type !kb:article)
(add-triple *demo-article* !kb:containsPerson
            !"Barack Obama")
(add-triple *demo-article* !kb:containsPerson
            !"Bill Clinton")
(add-triple *demo-article* !kb:containsPerson
            !"Bill Jones")

(print (freetext-get-ids "Clinton"))

;; get results using a cursor (recommended):
(iterate-cursor (triple (freetext-get-triples
                        '(and "Bill" "Jones")))
               (print triple))
(iterate-cursor (triple (freetext-get-triples "Bill"))
               (print triple))
(iterate-cursor (triple (freetext-get-triples
                        '(or "Jones" "Clinton")))
               (print triple))

;; get results as a list:
(print
 (freetext-get-triples-list '(or "Bill" "Barack")))

```

In this example I used the Lisp APIs for finding triples containing search terms. You will see in Chapter 5 how to use text search in SPARQL queries and in Chapter 9 I will show you how to use text search using Franz's Prolog query interface.

TBD: wrapup

1.3.4 Comparing AllegroGraph With Other Semantic Web Frameworks

Although Volume II of this book is about developing Semantic Web applications using AllegroGraph, it is also worthwhile looking at alternative technologies that can be used in addition to or instead of AllegroGraph. The two alternative technologies that I have used most for Semantic Web applications are Swi-Prolog with its Semantic Web libraries (open source, LGPL) and the Java Sesame project (open source, BSD style license). Swi-Prolog is an excellent tool for experimenting and learning about the Semantic Web. Sesame is a complete Java framework that is appropriate for applications written in Java. These alternatives have the advantage of being free

to use but lack advantages of scalability and utility that a commercial product like AllegroGraph has.

1.4 AllegroGraph Quickstart Wrap Up

This short chapter gave you a brief introduction to running AllegroGraph interactively and some of the APIs that you will be using most frequently. The next chapter shows you how to use Java, Ruby, Lisp, and Python clients with the standalone AllegroGraph server.

Part I

Semantic Web Technologies

2 Linked Data and the World Wide Web

It has been a decade since Tim Berners-Lee started writing about “version 2” of the world wide web: the Semantic Web. His new idea was to augment HTML anchor links with typed links using RDF data. As we will see in great detail throughout this book, RDF is encoded as data triples with the parts of each triple identified as the **subject**, **predicate**, and **object**. The **predicate** identifies the type of link between the **subject** and the **object** in a RDF triple.

The idea of linking data resources using RDF extends the web so that both human readers and software agents can use data resources.

2.1 Linked Data Resources on the Web

TBD

2.2 Graph Visualization of Linked Data using Gruff

Gruff is a commercial product based on AllegroGraph that provides an interactive graph browser for exploring RDF repositories. You might find it useful to load RDF triple data from examples in tis book into Gruff.

Gruff is available for Windows and Linux. A free version is available that is limited to 50 million RDF triples.

TBD: a screen shot

TBD

2.3 The Future: RDFa?

TBD: overview

TBD: example HTML+RDFa

TBD: Lisp code example for extracting RDF from HTML+RDFa

TBD

3 RDF

TBD: this chapter is really light on detail - fix.

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.

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.

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 data formats.

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.” There are many tools available that 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

A statement in RDF is a triple composed of a subject, predicate, and object.

While we tend to think in terms of objects and classes when using object oriented programming languages, we need to readjust our thinking when dealing with knowledge assets on the web. Instead of thinking about “objects” we deal with “resources” that are specified by URIs. In this way resources can be uniquely defined. We will soon see how we can associate different namespaces with URI prefixes – this will make it easier to deal different resources with the same name that can be found in different sources of information.

While subjects will almost always be represented as URIs of resources, the object part of triples can be either URIs of resources or literal values. For literal values, the XML schema notation for specifying either a standard type like integer or string, or a custom type that is application domain specific.

You have probably read articles and other books on the Semantic Web, and if so, you are probably used to seeing RDF expressed in its XML serialization format: you will not see XML serialization in this book. Much of my own confusion when I was starting to use Semantic Web technologies ten years ago was directly caused by trying to think about RDF in XML form. RDF data is graph data and serializing RDF as XML is confusing and a waste of time, really, when either the N-Triple format or even better, the N3 format are so much easier to read and understand.

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. I deal with triples like:

- subject: a URL (or URI) of a news article
- predicate: a relation like "containsPerson"
- object: a value like "Bill Clinton"

As previously mentioned, we will use either URIs or string literals as values for subjects and objects. We will always use URIs for the values of predicates. In any case URIs are usually preferred to string literals because they are unique. We will see an example of this preferred use but first we need to learn the N-Triple and N3 RDF formats.

3.1 RDF Examples in N-Triple and N3 Formats

In the Introduction 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. I will try to make this idea more clear with some examples.

Any part of a triple (subject, predicate, or object) is either a URI or a string literal. URIs encode namespaces. For example, the containsPerson property is used as the value of the predicate in this triple; the last example could properly be written as:

```
http://knowledgebooks.com/ontology/#containsPerson
```

The first part of this URI is considered to be the namespace for (what we will use as a predicate) “containsPerson.” Once we associate an abbreviation like **kb** for **http://knowledgebooks.com/ontology/** then we can just use the QName (“quick name”) with the namespace abbreviation; for example:

```
kb:containsPerson
```

Being able to define abbreviation prefixes for namespaces makes RDF and RDFS files shorter and easier to read.

When different RDF triples use this same predicate, this is some assurance to us that all users of this predicate subscribe to the same meaning. Furthermore, we will see in Section 4.1 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 a predicate like “containsPerson” in the way that a human reader can by combining understood common meanings for the words “contains” and “person” but for many interesting and useful types of applications that is fine as long as the predicate is used consistently.

Because there are many sources of information about different resources the ability to define different namespaces and associate them with unique URI prefixes makes it easier to deal with situations

A statement in N-Triple format consists of three URIs (or string literals – any combination) 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. I often use either command line tools or the Java Sesame library to convert N-Triple files to N3 if I will be reading them or even hand editing them. You will see why I prefer the N3 format when we look at an example:

```
@prefix kb: <http://knowledgebooks.com/ontology#> .  
<http://news.com/201234 /> kb:containsCountry "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. When we want to use a URL as a URI, we enclose it in angle brackets – as in this example. 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.”

3 RDF

This was a very simple N3 example which we will expand to show additional features of the N3 notation. As another example, suppose that this news article also mentions the USA. Instead of adding a whole new statement like this:

```
@prefix kb: <http://knowledgebooks.com/ontology#> .
<http://news.com/201234 /> kb:containsCountry "China" .
<http://news.com/201234 /> kb:containsCountry "USA" .
```

we can combine them using N3 notation. N3 allows us to 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 "China" ,
                                "USA" .
```

We can also add in additional predicates that use the same subject:

```
@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 we use (we will be using AllegroGraph) it makes no difference if we load RDF as XML, N-Triple, or N3 format files: internally subject, predicate, and object triples are stored in the same way and are used in the same way.

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 the N-Triple format:

```
<http://news.com/2034 /> kb:datePublished "2008-05-11" .
```

Furthermore, if we do not have dates for all news articles that is often acceptable depending on the application.

3.2 The RDF Namespace

You saw a few examples of using namespaces in Chapter 1 where I registered my own namespace `http://knowledgebooks.com/rdfs#` and used the AllegroGraph function (**display-namespaces**) to display all available namespaces in the currently opened AllegroGraph data store.

When you register a name space you can assign any “Quick name” (QName, or abbreviation) to the URI that uniquely identifies a namespace.

The RDF namespace `http://www.w3.org/1999/02/22-rdf-syntax-ns#` is usually registered with the QName **rdf:** and I will use this convention. The next few sections show the definitions in the RDF namespace that I use in this book.

3.2.1 rdf:type

The **rdf:type** property is used to specify the type (or class) of a resource. Notice that we do not capitalize “type” because by convention we do not capitalize RDF property names. Using an example from Chapter 1, but in more detail:

```
(defvar *demo-article*
  (resource
    "http://demo_news/12931"))
(add-triple *demo-article*
  !rdf:type
  !kb:article)
```

Here we are converting the URL of a news web page to a resource and then defining a new triple that specifies the web page resource is of type `kb:article` (again, using the QName **kb:** for my knowledgebooks.com namespace).

3.2.2 rdf:Property

The **rdf:Property** class is, as you might guess from its name, used to describe and define properties. Notice the “Property” is capitalized because by convention we capitalize RDF class names.

This is a good place to show how we define new properties, using a previous example:

```
(add-triple *demo-article*
  !kb:containsPerson
  !"Barack Obama")
```

The **kb:containsPerson** property might be defined using:

```
(add-triple !kb:containsPerson
  !rdf:type
  !rdf:Property)
```

Here I am using the AllegroGraph APIs for defining triples programmatically; the equivalent definition in N3format is:

```
kb:containsPerson rdf:type rdf:Property .
```

When we discuss RDF Schema (RDFS) in Chapter 4 we will see how to create subtypes and sub-properties.

3.3 RDF Wrap Up

If you read the World Wide Web Consortium's RDF Primer (highly recommended) at <http://www.w3.org/TR/REC-rdf-syntax/> you will see many other classes and properties defined that in my opinion are often most useful when dealing with XML serialization of RDF. Using the N-Triple and N3 formats, I find that I usually just use `rdf:type` and `rdf:Property` in my own modeling efforts, along with a few identifiers defined in the RDFS namespace that we will look at in the next chapter.

An RDF triple has three parts: a subject, predicate, and object. In later chapters we will see that AllegroGraph also stores a unique integer triple ID and a graph ID (for partitioning RDF data and to support graph operations). We will look at these extensions in some detail in Chapter ???. While using the triple ID and graph ID can be useful, my own preference is to stick with using just what is in the RDF standard.

By itself, RDF is good for storing and accessing data but lacks functionality for modeling classes, defining properties, etc. We will extend RDF with RDF Schema (RDFS) in the next chapter.

4 RDFS

The World Wide Web Consortium RDF Schema (RDFS) definition can be read at <http://www.w3.org/TR/rdf-schema/> and I recommend that you use this as a reference because I will only discuss the parts of RDFS that are required for implementing the examples in this book. The RDFS namespace <http://www.w3.org/2000/01/rdf-schema#> is usually registered with the QName **rdf:** and I will use this convention.

4.1 Extending RDF with RDF Schema

RDFS supports the definition of classes and properties based on set inclusion. In RDFS classes and properties are orthogonal. We will not simply be using properties to define data attributes for classes – this is different than object modeling and object oriented programming languages like Java. RDFS is encoded as RDF – the same syntax.

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. Understanding the difficulty of integrating different data sources in different formats helps to understand the design decisions behind the Semantic Web.

We will start with an example that is an extension of the example in the last section that also uses RDFS. After defining **kb:** and **rdfs:** namespace QNames, we add a few additional RDF statements (that are RDFS):

```
@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 (that are themselves valid RDF triples) declare that:

- The property `containsCity` is a subproperty of `containsPlace`.
- The property `containsCountry` is a subproperty of `containsPlace`.
- The property `containsState` is a subproperty 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 property 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 subproperty 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.

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 more RDFS features in Section ?? to perform SPARQL queries.

4.2 Modeling with RDFS

While RDFS is not as expressive of a modeling language as the RDFS++ and OWL languages that we will cover in Chapter 6, the combination of RDF and RDFS is likely adequate for many semantic web applications. Reasoning with and using more expressive modeling languages will require increasingly more processing time. Combined with the simplicity of RDF and RDFS it is a good idea to start with less expressive and only “move up the expressivity scale” as needed.

Here is a short example on using RDFS to extend RDF (assume that my namespace **kb:** and the RDFS namespace **rdfs:** are defined):

```
kb:Person rdf:type rdfs:Class .
kb:Person rdfs:comment "represents a human" .
kb:Manager rdf:type kb:Person .
kb:Manager rdfs:domain kb:Person .
kb:Engineer rdf:type kb:Person .
kb:Engineer rdfs:domain kb:Person .
```

Here we see the use of **rdfs:comment** used to add a human readable comment

to the new class **kb:Person**. When we define the new classes **kb:Manager** and **kb:Engineer** we make them subclasses of **kb:Person** instead of the top level super class **rdfs:Class**. We will look at detailed interactive examples in Chapter 7 that demonstrate the utility of models using class hierarchies and hierarchies of properties – for now it is enough to introduce the notation.

The **rdfs:domain** of an **rdf:property** specifies the class of the subject in a triple while **rdfs:range** of an **rdf:property** specifies the class of the object in a triple. Just as strongly typed programming languages like Java help catch errors by performing type analysis, creating (or using existing) good RDFS property and class definitions helps RDFS, RDFS++, and OWL descriptive logic reasoners to catch modeling and data definition errors. These definitions also help reasoning systems infer new triples that are not explicitly defined in a triple data store.

We continue the current example by adding property definitions and then asserting a triple that is valid given the type and property restrictions that we have defined using RDFS:

```
kb:supervisorOf rdfs:domain kb:Manager .
kb:supervisorOf rdfs:range kb:Engineer .

"Mary Johnson" rdf:type kb:Manager .
"John Smith" rdf:type kb:Engineer .

"Mary Johnson" kb:supervisorOf "John Smith" .
```

If I tried to add a triple with “Mary Johnson” and “John Smith” reversed in the last RFD statement then an RDFS inference/reasoning system could catch the error. This example is not ideal because I am using string literals as the subjects in triples. In general, you probably want to define a specific namespace for concrete resources representing entities like the people in this example.

The property **rdfs:subClassOf** is used to state that all instances of one class are also instances of another class. The property **rdfs:subPropertyOf** is used to state that all resources related by one property are also related by another; for example given the following N3 statements that use string literals as resources to make this example shorter:

```
kb:familyMember rdf:type rdf:Property .
kb:ancestorOf rdf:type rdf:Property .
kb:parentOf rdf:type rdf:Property .

kb:ancestorOf rdfs:subPropertyOf kb:familyMember .
kb:parentOf rdfs:subPropertyOf kb:ancestorOf .
```

4 RDFS

```
"Marry Smith" kb:parentOf "Sam" .
```

then the following is valid:

```
"Marry Smith" kb:ancestorOf "Sam" .  
"Marry Smith" kb:familyMember "Sam" .
```

We have just seen that a common use of RDFS is to define additional application or data-source specific properties and classes in order to express relationships between resources and the types of resources. Whenever possible you will want to reuse existing RDFS properties and resources that you find on the web. For example, in the last example I defined my own subclass **kb:person** instead of using the Friend of a Friend (FOAF) namespace's definition of person.

Modeling and reasoning (inferencing) are tightly coupled and we will wait until our discussion of RDFS++ in Chapter 6 and inferencing in Chapter 7 to experiment with the AllegroGraph reasoning and querying system.

4.3 TBox and ABox

TBD

4.4 Descriptive Logic Classes are Derivable

Class membership in Descriptive Logic is derivable: we define constraints for class membership, then membership in a class can be computed. This augments explicit statements of class membership.

TBD

4.5 AllegroGraph RDFS++ Extensions

TBD

4.6 RDFS Wrapup

5 The SPARQL Query Language

SPARQL is a query language used to query RDF data stores. While SPARQL may initially look like SQL you will see that there are some important differences like support for RDFS and OWL inferencing (see Chapter 7) and graph-based instead of relational matching operations. We will cover the basics of SPARQL in this section and then see more examples in Chapter 7 when we learn about the AllegroGraph query APIs.

5.1 Example RDF Data in N3 Format

We will use the N3 format RDF file `chapter_07_sparql/news.n3` for examples in this chapter. We use the N3 format because it is easier to read and understand. In Chapter 7 we will use the equivalent N-Triple format file `chapter_07_sparql/news.nt` because AllegroGraph does not currently support loading N3 files. These files were created automatically by spidering Reuters news stories on the `news.yahoo.com` web site and automatically extracting named entities from the text of the articles. I used the Java Sesame library to convert the generated N-Triple file to N3 format. We will see techniques for extracting named entities from text in Chapter 10 when I develop utilities for using the Reuters Open Calais web services. In this chapter we use these sample RDF files that I have created using Open Calais and news articles that I found on the web.

You have already seen snippets of this file in Section 4.1 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" .

```

5.2 Example SPARQL SELECT Queries

In the following examples, we will look at queries but not the results. Please be patient: these same queries are used in the interactive AllegroGraph examples in later chapters so it makes sense to only list the query return values in one place. Besides that, you will enjoy running the example programs yourself and experiment with modifying the 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*:

```
SELECT ?subject ?object
WHERE {
  ?subject
  http://knowledgebooks.com/ontology#containsCountry>
  ?object .
}
```

Variables in queries start with a question mark character and can have any names. We can make this query easier 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 .
}
```

We could have filtered on any other predicate, for instance **containsPlace**. Here is another example using a match against a string literal to find all articles exactly matching the text “Maryland.” The following queries were copied from Java source files and were embedded as string literals so you will see quotation marks backslash escaped in these examples. If you were entering these queries into a query form you would not escape the quotation marks.

```
PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject
WHERE { ?subject kb:containsState \"Maryland\" . }
```

We can also match partial string literals against regular expressions:

```
PREFIX kb:
```

```

SELECT ?subject ?object
  WHERE {
    ?subject
    kb:containsOrganization
    ?object FILTER regex(?object, \"University\") .
  }

```

Prior to this last example query we only requested that the query return values for subject and predicate for triples that matched the query. However, we might want to return all triples whose subject (in this case a news article URI) is in one of the matched triples. Note that there are two matching triples, each terminated with a period:

```

PREFIX kb: <http://knowledgebooks.com/ontology#>
SELECT ?subject ?a_predicate ?an_object
  WHERE {
    ?subject
    kb:containsOrganization
    ?object FILTER regex(?object, \"University\") .

    ?subject ?a_predicate ?an_object .
  }
DISTINCT
ORDER BY ?a_predicate ?an_object
LIMIT 10
OFFSET 5

```

When WHERE clauses contain more than one triple pattern to match, this is equivalent to a Boolean “and” operation. The DISTINCT clause removes duplicate results. The ORDER BY clause sorts the output in alphabetical order: in this case first by predicate (containsCity, containsCountry, etc.) and then by object. The LIMIT modifier limits the number of results returned and the OFFSET modifier sets the number of matching results to skip.

We are done with our quick tutorial on using the SELECT query form. There are three other query forms that I am not covering in this chapter:

- CONSTRUCT – returns a new RDF graph of query results
- ASK – returns Boolean true or false indicating if a query matches any triples
- DESCRIBE – returns a new RDF graph containing matched resources

5.3 Example SPARQL CONSTRUCT Queries

TBD

5.4 Example SPARQL ASK Queries

TBD

5.5 Example SPARQL DESCRIBE Queries

TBD

This chapter ends the background material on Semantic Web Technologies. The remaining chapters in this book will be Common Lisp and AllegroGraph specific.

6 RDFS++ and OWL

There are three standard versions of OWL: Lite, Description Logic (DL), and Full.

Because more expressive versions of OWL require computing resources (and OWL Full may often not be useful except for small problems) it is usually a good idea to use the minimum modeling constructs when developing a Semantic Web application.

OWL DL strikes a good balance between expressiveness and computability. OWL DL reasoners are usually complete (that is, they provide all possible answers to queries). The problem in many real-world applications is the unpredictability of how long a query will take to execute.

While the three versions of OWL are standards there is an ad-hoc definition called RDFS++ that is more expressive than RDF + RDFS but less expressive than OWL. Since AllegroGraph supports RDFS++ but not OWL (without using external reasoning systems) we will not cover OWL constructs in this book unless they are implemented in RDFS++.

6.1 Properties Supported In RDFS++

The *unofficial* version of RDFS/OWL called RDFS++ is a practical compromise between DL OWL and RDFS inferencing. AllegroGraph supports the following predicates:

- `rdf:type` – discussed in Chapter 3
- `rdf:property` – discussed in Chapter 3
- `rdfs:subClassOf` – discussed in Chapter 4
- `rdfs:range` – discussed in Chapter 4
- `rdfs:domain` – discussed in Chapter 4
- `rdfs:subPropertyOf` – discussed in Chapter 4
- `owl:sameAs`
- `owl:inverseOf`
- `owl:TransitiveProperty`

We will now discuss **owl:sameAs**, **owl:inverseOf**, and **owl:TransitiveProperty** to complete the discussion of frequently used RDFS predicates in Chapter 4. We will see in Chapters 7 and 8 interactive examples of these predicates.

6.1.1 owl:sameAs

If the same entity is represented by two distinct URIs **owl:sameAs** can be used to assert that the URIs refer to the same entity. For example, two different knowledge sources might define different URIs in their own namespaces for President Barack Obama. Rather than translate data from one knowledge source to another it is simpler to equate the two unique URIs. For example:

```
kb:BillClinton rdf:type kb:Person .
kb:BillClinton owl:sameAs mynews:WilliamClinton
```

Then the following can be verified using a RDFS++ or OWL DL capable reasoner:

```
mynews:WilliamClinton rdf:type kb:Person .
```

6.1.2 owl:inverseOf

We can use **owl:inverseOf** to declare that one property is the inverse of another.

```
:parentOf owl:inverseOf :childOf .
"John Smith" :parentOf "Nellie Smith" .
```

There s something new in this example: I am using a “default namespace” for **:parentOf** and **:childOf**. A default namespace is assumed to be application specific and that no external software agents will need access to resources defined in the default namespace.

Given the two previous RDF statements we can infer that the following is also true:

```
"Nellie Smith" :childOf "John Smith" .
```

6.1.3 owl:TransitiveProperty

As its name implies **owl:TransitiveProperty** is used to declare that a property is transitive as the following example shows

```
kb:ancestorOf \textbf{a} rdf:Property .  
"John Smith" kb:ancestorOf "Nellie Smith" .  
"Nellie Smith" kb:ancestorOf "Billie Smith" .
```

There is something new in this example: in N3 you can use **a** as shorthand for **rdf:type**. Given the last three RDF statements we can infer that:

```
"John Smith" : kb:ancestorOf "Billie Smith" .
```

6.2 RDF, RDFS, and RDFS++ Modeling Wrap Up

You should now be getting the idea that RDF, RDFS, and RDFS++ Modeling is quite different than object modeling in object oriented software development

TBD

TBD

Part II

AllegroGraph Extended Tutorial

7 SPARQL Queries Using AllegroGraph APIs

We saw some example SPARQL queries in Chapter 5 where we expressed the queries in text form. In this chapter we will work through SPARQL examples using snippets of Lisp code and the AllegroGraph APIs. We will see more interactive examples that are built on the examples in this chapter when we look at more reasoning examples in Chapter 8 and AllegroGraph's Prolog interface in Chapter 9.

The file `chapter_07_sparql/news.nt` was generated automatically by spidering a list of Reuters news articles on Yahoo News and using the Open Calais entity extraction web services that we will discuss in some detail in Chapter 10. This generated N-Triple file does not use name space abbreviations as you can see from the first line in the file:

```
<http://news.yahoo.com/20080616/ts_nm/usa_flooding_dc_16 />  
<http://knowledgebooks.com/ontology#containsCity>  
  "Burlington" .
```

7.1 Using Namespaces

We have seen in earlier chapters how we use RDF triples from different sources that we identify as belonging to different namespaces. The function **register-namespace** is used to associate quick name abbreviations with namespace URIs.

AllegroGraph does not support reading the concise N3 format that we used in the last chapter but we can make the N-Triple data file easier to work with by copying it to the file `news_ns.nt` and using edit macros to convert to using the namespaces:

```
(register-namespace  
  "kb"  
  "http://knowledgebooks.com/ontology#")  
(register-namespace  
  "test_news"  
  "http://news.yahoo.com/s/nm/20080616/ts_nm")
```

7.2 Reading RDF Data From Files

We created new RDF triples programatically in Chapter 1. In this section we will see how to read triple stores from disk.

In the last section we registered the namespace **test_news** that is used in the first line of the edited file `news_ns.nt`:

```
!test_news:usa_flooding_dc_16
!kb:containsCity
"Burlington" .
```

The file `chapter_07_sparql/sparql_query.lisp` contains all of the examples in this chapter. I encourage you to read through this chapter as well as the next two AllegroGraph tutorial chapters with a Lisp listener window open and try all examples for yourself and then experiment with the techniques we cover in the text.

We need to load this N-Triple file before performing and queries:

```
(load-ntriples "news.nt")
```

This load operation will fail if you have not defined the namespaces used in the file with the function **register-namespace**.

7.3 Lisp APIs for Queires

If we try a query the default is to return RDF in XML format (and we agreed to not use XML encodings!) If you are following this tutorial interactively, try evaluating the following expression:

```
(sparql:run-sparql "
  PREFIX kb: <http://knowledgebooks.com/ontology#>
  SELECT ?article_uri ?city_name WHERE {
    ?article_uri kb:containsCity ?city_name .
  }")
```

Fortunately we can use optional arguments on the **sparql:run-sparql** function to get a more convenient “Lisp like” return values for SPARQL queries. This example gets the results as a list of hash tables:

```
(defvar *r1*
  (sparql:run-sparql "
    PREFIX kb: <http://knowledgebooks.com/ontology#>
    SELECT ?article_uri ?city_name WHERE {
      ?article_uri kb:containsCity ?city_name .
    }"
    :results-format :hashes))

(dolist (result *r1*)
  (maphash
    #'(lambda (key value)
        (format t " key: ~S~% value: ~S~%~%"
              key value))
    result))
```

The output from this code snippet is:

```
key: |?article_uri|
value: {http://news.yahoo.com/...}

key: |?city_name|
value: {Burlington}
;; etc.
```

The SPARQL **ASK** command checks to see if a given query produces any results. The following example request a Lisp true/false return value to the question “Does any article contain the city Chicago?”:

```
(sparql:run-sparql "
  PREFIX kb: <http://knowledgebooks.com/ontology#>
  ASK {
    ?any_article kb:containsCity 'Chicago'
  }"
  :results-format :boolean)
```

There are many possible options for the **:results-format** keyword argument, including:

- `:sparql-xml` – serializes the results as XML to output-stream
- `:sparql-json` – serializes the results as JSON data to output-stream
- `:sparql-ttl` – serializes the results as Turtle encoding to output-stream (Turtle is a simplified version of N3, like N-Triples with namespaces)

- `:hashes` – returns a list of hash tables (as seen in a previous example)
- `:arrays` – returns a list of arrays for each results
- `:lists` – returns a list of lists for each results
- `:count` – returns an integer for the number of results

At some loss of efficiency it is sometimes useful to match string values against regular expressions; for example:

```
(sparql:run-sparql "  
  PREFIX kb: <http://knowledgebooks.com/ontology#>  
  SELECT ?article_uri WHERE {  
    ?article_uri kb:containsPerson ?person_name .  
    FILTER regex(?person_name, '^*Putin*')  
  }"  
  :results-format :lists)  
  
;; output:  
(((http://news.yahoo.com/s/nm/20080616/ts_nm/worldleaders /)))
```

7.4 Wrap Up

You learned how to query RDF triples in a repository in this chapter. We only considered triples that we explicitly added to the triple store. However much of the power of Semantic Web technologies in general and AllegroGraph in particular is the ability to use triples that are inferred from RDFS without being explicitly created. This capability is covered in the next chapter in addition to techniques for using different data sources implemented using different schemas.

8 AllegroGraph Reasoning System

In the last chapter we saw how SPARQL queries can be used to find specific data in an RDF graph. So far we have only seen examples of finding data that has been explicitly added to an RDF data repository.

However RDFS, RDFS++, and OWL reasoners can return results that are know implicitly by using inference. We saw in Chapter 6 that AllegroGraph supports reasoning using the following predicates:

- `rdf:type` – discussed in Chapter 3
- `rdf:property` – discussed in Chapter 3
- `rdfs:subClassOf` – discussed in Chapter 4
- `rdfs:range` – discussed in Chapter 4
- `rdfs:domain` – discussed in Chapter 4
- `rdfs:subPropertyOf` – discussed in Chapter 4
- `owl:sameAs` – discussed in Chapter 6
- `owl:inverseOf` – discussed in Chapter 6
- `owl:TransitiveProperty` – discussed in Chapter 6

8.1 Enabling RDFS++ Reasoning on a Triple Store

We will look at the AllegroGraphs APIs and programming techniques for reasoning in detail in this chapter. By default AllegroGraph triple stores do not support RDFS++ reasoning. You must enable RDFS++ reasoning functionality by:

```
(apply-rdfs++-reasoner :db *db*)
```

This function works via side effect: the specified data store is converted to support inferencing. Since the default database `*db*` can be assumed, this can be shortened to:

```
(apply-rdfs++-reasoner)
```

If you use multiple data stores at the same time you can use different inference support for each.

TBD ...

8.2 Inferring New Triples: `rdf:type` vs. `rdfs:subClassOf` Example

Implementations of RDF triple stores that support RDFS, RDFS++, or OWL reasoning can implement inferred triples in different ways. One approach is to “pre-calculate” inferred triples using forward chaining inference. A different approach is to infer triples at query time. The results should (hopefully) be the same.

In the following example, we define two triples and then perform a SPARQL query that answers a question based on a triple that has been explicitly added to the triple store:

```
(add-triple !kb:man !rdfs:type !kb:person)
(add-triple !kb:sam !rdf:type !kb:man)

(sparql:run-sparql "
  PREFIX kb: <http://knowledgebooks.com/ontology#>
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  ASK {
    kb:sam rdf:type kb:man
  }"
  :results-format :boolean)
```

This query returns a Lisp true value **T**. You might think that since **kb:man** is declared of **rdf:type kb:person** that the following query would return a true value:

```
(add-triple !kb:man !rdf:type !kb:person)
(add-triple !kb:sam !rdf:type !kb:man)

(sparql:run-sparql "
  PREFIX kb: <http://knowledgebooks.com/ontology#>
  PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
  ASK {
    kb:sam rdf:type kb:person
```

```
}"  
:results-format :boolean)
```

This, however, returns a Lisp false value **NIL**. To get what you probably thought was the expected subclass behavior we can use **rdfs:subClassOf**:

```
(add-triple !kb:man !rdfs:subClassOf !kb:person)
```

Now the last query returns a true (Lisp **T**) value.

TBD

TBD

8.3 Controlling AllegroGraph's Inferencing Engine

TBD

9 AllegroGraph Prolog Interface

TBD

Part III

Common Lisp Utilities for Information Processing

10 Common Lisp Client Library for Open Calais

The Open Calais web services are available for free use with some minor limitations. This service is also available for a fee with additional functionality and guaranteed service levels. We will use the free service in this chapter.

TBD: give a better plug for Reuters here, including a URL

You will need to apply for a free developers key. On my development systems I define an environment variable for the value of my key using (the key shown is not a valid key, by the way):

```
export OPEN_CALAIS_KEY=po2eq112hkf985f3k
```

The example source files are found in `lisp_practical_semantic_web/opencalais`:

- `load.lisp` – loads and runs the demo
- `opencalais-lib.lisp` – performs web service calls to find named entities in text
- `opencalais-data-store.lisp` – maintains an RDF data store for named entities
- `test-opencalais.lisp` – demo test program

10.1 Open Calais Web Services Client

The Open Calais web services return RDF payloads serialized as XML data.

TBD: describe full schemas used

For our purposes, we will not use the returned XML data and instead parse the comment block to extract named entities that Open Calais identifies. There is a possibility in the future that the library in this section may need modification if the format of this comment block changes (it has not changed in several years).

I will not list all of the code in `opencalais-lib.lisp` but we will look at some of it. I start by defining two constant values, the first depends on your setting of the environment variable `OPEN_CALAIS_KEY`:

```
(defvar *my-opencalais-key* (sys::getenv "OPEN_CALAIS_KEY"))

(defvar *PARAMS*
  (concatenate 'string
    "&paramsXML="
    (MAKE-ESCAPED-STRING
      "<c:params ... >.....</c:params>"))) )
```

The web services client function is fairly trivial: we just need to make a RESTful web services call and extract the text from the comment block, parsing out the named entities and their values. Before we look at some code, we will jump ahead and look at an example comment block; understanding the input data will make the code easier to follow:

```
<!--Relations: PersonCommunication,
              PersonPolitical,
              PersonTravel

Company: IBM, Pepsi
Country: France
Person: Hiliary Clinton, John Smith
ProvinceOrState: California-->
```

We will use the **net.aserve.client:do-http-request** function to make the web service call after setting up the RESTful arguments:

```
(defun entities-from-opencalais-query (query
  &aux url results index1 index2
  lines tokens hash)
  (setf hash (make-hash-table :test #'equal))
  (setf url
    (concatenate 'string
      "http://api.opencalais.com/enlighten"
      "/calais.asmx/Enlighten?"
      "licenseID="
      *my-opencalais-key*
      "&content="
      (MAKE-ESCAPED-STRING query)
      *PARAMS*))
  (setf results (net.aserve.client:do-http-request url))
```

The value of results will be URL-encoded text and for our purposes there is no need to decode the text returned from the web service call:

```
(setq index1 (search "terms of service.--&gt;" results))
(setq index1 (search "&lt;!--" results :start2 index1))
(setq index2 (search "--&gt;" results :start2 index1))
(setq results (subseq results (+ index1 7) index2))
(setq lines
  (split-sequence:split-sequence #\Newline results))
(dolist (line lines)
  (setq index1 (search ":" line))
  (if index1
      (let ((key (subseq line 0 index1))
            (values (split-sequence:split-sequence ", "
              (subseq line (+ index1 2))))))
        (if (not (string-equal "Relations" key))
            (setf (gethash key hash) values))))))
(maphash
 #'(lambda (key val)
     (format t "key: ~S val: ~S~%" key val))
 hash)
hash)
```

Before using this utility function in the next section to fetch data for an RDF data store we will look at a simple test:

```
(entities-from-opencalais-query
 "Senator Hiliary Clinton spoke with the president
 of France. Clinton and John Smith talked on
 the aiplane going to California. IBM and Pepsi
 contributed to Clinton's campaign.")
```

The debug printout in this call is:

```
key: "Country" val: ("France")
key: "Person" val: ("Hiliary Clinton" "John Smith")
key: "Company" val: ("IBM" "Pepsi")
key: "ProvinceOrState" val: ("California")
```

10.2 Storing Entity Data in an RDF Data Store

We will use the utilities developed in the last section for using the Open Calais web services in this section to populate an RDF data store. You can find the utilities developed in this section in the source file `opencalais-data-store.lisp`. We start by

making sure that the AllegroGraph libraries are loaded and we define a namespace that we will use for examples in the rest of this chapter:

```
;; Use the opencalais-lib.lisp utilities to create
;; an RDF data store. Assume that a AG RDF
;; repository is open.

(require :agraph)
(in-package :db.agraph.user)

(register-namespace
  "kb"
  "http://knowledgebooks.com/rdfs#")
```

To avoid defining a global variable for a hash table we define one locally inside a closure that also defines the only function that needs read access to this hash table:

```
(let ((hash (make-hash-table :test #'equal)))
  (setf (gethash "Country" hash) !kb:containsCountry)
  (setf (gethash "Person" hash) !kb:containsPerson)
  (setf (gethash "Company" hash) !kb:containsCompany)
  (setf (gethash "ProvinceOrState" hash)
        !kb:containsState)
  (setf (gethash "Product" hash) !kb:containsProduct)
  ;; utility function for getting a URI for a
  ;; predicate name:
  (defun get-rdf-predicate-from-entity-type (entity-type)
    (let ((et (gethash entity-type hash)))
      (if (not et)
          (progn
            ;; just use a string literal if there is
            ;; no entry in the hash table:
            (setf et entity-type)
            (format t
                    "Warning: entity-type ~S not defined
                    in opencalais-data-store.lisp~%"
                    entity-type)))
          et)))
```

Function **get-rdf-predicate-from-entity-type** is used to map string literals to specific predicates defined in the knowledgebooks.com namespace. The following function is the utility for processing the text from documents and generating multiple triples that all have their subject equal to the value of the unique URI for the original document.

```
(defun add-entities-to-rdf-store (subject-uri text)
  "subject-uri if the subject for triples that this
  function defines"
  (maphash
    #'(lambda (key val)
        (dolist (entity-val val)
          (add-triple
            subject-uri
            (get-rdf-predicate-from-entity-type key)
            (literal entity-val))))
    (entities-from-opencalais-query text)))
```

If documents are not plain text (for example a word processing file or a HTML web page) then applications using the utility code developed in this chapter need to extract plain text. The code in this section is intended to give you ideas for your own applications; you would at least substitute your own namespace(s) for your application.

10.3 Testing the Open Calais Demo System

The source file `test-opencalais.lisp` contains the examples for this section:

```
(require :agraph)
(in-package :db.agraph.user)

(create-triple-store "/tmp/rdfstore_1")
```

We start by using the utility function defined in the last section to find all named entities in sample text and create triples in the data store:

```
(add-entities-to-rdf-store
  !<http://newsdemo.com/1234>
  "Senator Hiliary Clinton spoke with the president
  of France. Clinton and John Smith talked on the
  aiplane going to California. IBM and Pepsi
  contributed to Clinton's campaign.")
```

We can print all triples in the data store:

```
(print-triples (get-triples-list) :format :concise)
```

and output is:

```
<1: http://newsdemo.com/1234 kb:containsCountry France>
<2: http://newsdemo.com/1234 kb:containsPerson
    Hiliary Clinton>
<3: http://newsdemo.com/1234 kb:containsPerson
    John Smith>
<4: http://newsdemo.com/1234 kb:containsCompany IBM>
<5: http://newsdemo.com/1234 kb:containsCompany Pepsi>
<6: http://newsdemo.com/1234 kb:containsState California>
```

This example showed just adding triples generated from a single document. If a large number of documents are processed then queries like the following might be useful:

```
(print-triples
 (get-triples-list
  :p (get-rdf-predicate-from-entity-type "Company")
  :o (literal "Pepsi"))
 :format :concise)
```

producing this output:

```
<5: http://newsdemo.com/1234 kb:containsCompany Pepsi>
```

Here we identify all documents that mention a specific company.

10.4 Open Calais Wrap Up

Since AllegroGraph supports indexing and search of any text fields in triples, the combination of using triples to store specific entities in a large document collection with full search, AllegroGraph can be an effective tool to manage large document repositories.

“Documents” can be any source of text identified with a unique URI: web pages, word processing documents, blog entries, etc.

We will use the utilities developed in this chapter when we design and build a web portal in Chapters 18, 19, and 20.

11 Common Lisp Client Library for Natural Language Processing

We used the Open Calais web services in Chapter 10 to identify proper names in text and we used this information to create RDF data linking information sources with people and places mentioned in the text. I will introduce you to my own library for natural language processing (NLP) in this chapter.

11.1 KnowledgeBooks.com Natural Language Processing Library

I started developing the KnowledgeBooks.com NLP library as a commercial product in the 1990s and continued to improve it until about 2008 when I started using the Open Calais web services for most of my own work and research. While the Open Calais system produces more accurate results than my own library, I still find it useful to use my own library so I took some effort to clean up my old code (removing a great deal of it, leaving the parts that you may find most useful) for inclusion in the software distribution for this book.

I will give you a quick tutorial in this chapter on using the major APIs in my library. If you are interested in the code itself then you can browse through the code.

11.2 Entity Extraction from Text

TBD

11.2.1 Entity Extraction from Text Library

TBD: use a simplified version of my code, discuss APIs, etc.

11.2.2 Entity Extraction with AllegroGraph Example

TBD: like the later Open Calais chapter, put entities in a data store, demo queries, etc.

11.3 Automatic Text Tagging

TBD

11.3.1 KnowledgeBooks.com Text Tagging Library

TBD: use a simplified version of my code, discuss APIs, etc.

11.3.2 Text Tagging with AllegroGraph Example

TBD: like the later Open Calais chapter, put entities in a data store, demo queries, etc.

11.4 Automatically Summarizing Text

TBD

11.4.1 KnowledgeBooks.com Automatic Summarization Library

TBD: use a simplified version of my code, discuss APIs, etc.

11.4.2 Automatic Summarization and AllegroGraph Index/Search Example

TBD

12 Common Lisp Client Library for Freebase

Freebase is a public data source created by the MetaWeb Corporation. Freebase is similar to Wikipedia because uses add data that is linked to other data in Freebase. If you are not already familiar with Freebase then I suggest you spend some time experimenting with the web interface (<http://freebase.com>) before working through this chapter. As a developer make sure that you eventually look at the developer documentation at <http://www.freebase.com/docs/data> because I will only the aspects of Freebase that I need for the example applications in this book.

12.1 Overview of Freebase

TBD

12.1.1 MQL Query Language

TBD

12.1.2 Search

TBD

12.1.3 Geo Search

TBD

12.2 Freebase Common Lisp Client Library

TBD

13 Common Lisp Client Library for DBpedia

DBpedia is a mostly automatic extraction of RDF data from Wikipedia using the metadata in Wikipedia articles. You have two alternatives for using DBpedia in your own applications: using the public DBpedia SPARQL endpoint or downloading all or part of the DBpedia RDF data and loading it into your own RDF data store (e.g., AllegroGraph or Sesame).

The public DBpedia SPARQL endpoint URI is <http://dbpedia.org/sparql>. For the purpose of the examples in this book we will simply use the public SPARQL endpoint but for serious applications I suggest that you run your own endpoint.

TBD

13.1 Using the AllegroGraph SPARQL Client Library to access DBpedia

TBD

13.2 Example SPARQL Queries Using DBpedia

TBD

14 Common Lisp Client Library for Exporting SPARQL Query Results

TBD

14.1 Exporting to OpenOffice.org Spreadsheets

TBD

14.2 Exporting to PostgreSQL Databases

TBD

Part IV

Sample Application: KnowledgeBooks Semantic Web Spider

15 Semantic Web Spider: Information Gathering

TBD

16 Semantic Web Spider: RDF Data Storage

TBD

17 Semantic Web Spider: APIs

TBD

Part V

Sample Application: Semantic Web Enabled Web Site

18 Requirements and Design of the Semantic Web Portal Web Application

The example application developed in this part of the book is a Semantic Web Portal Web Application with the following features:

- Indexes content from the web and local text documents.
- Web application interface for search. Search results include links to other similar documents that are associated by category, people and places discussed in the documents, and clustered using K-Means.
- Repository and indexes are also searchable as a SPARQL endpoint.

TBD

19 Web Application Back End Implementation

TBD

20 Web Interface for the Semantic Web Portal

TBD

20.1 Introduction to AllegroServe

TBD

20.2 Introduction to Web Actions

TBD

20.3 Dojo and Javascript for the Web Interface

TBD

20.4 Web Application Implementation

TBD

Bibliography