

Practical Semantic Web Programming With AllegroGraph

Mark Watson

Copyright 2008 Mark Watson. All rights reserved.
This work is licensed under a Creative Commons
Attribution-Noncommercial-No Derivative Works
Version 3.0 United States License.

November 24, 2008

Contents

Preface	xi
1. Introduction	1
1.1. Who is is this book written for?	1
1.2. Why is a PDF copy of this book available free on the web?	1
1.3. Book Software	2
1.4. Why Graph Data Representations are Better than the Relational Database Model for Dealing with Rapidly Changing Data Requirements	3
1.5. Book Summary	3
2. AllegroGraph Quick Start	5
2.1. Starting AllegroGraph	5
2.2. Working with RDF Data Stores	6
2.2.1. Creating Repositories	6
2.2.2. AllegroGraph Lisp Reader Support for RDF	7
2.2.3. Adding Triples	8
2.2.4. Saving Triple Stores to Disk as XML, N-Triples, and N3	10
2.3. AllegroGraph Quickstart Wrap Up	11
I. Semantic Web Technologies	13
3. RDF	15
3.1. RDF Examples in N-Triple and N3 Formats	16
3.2. The RDF Namespace	18
3.2.1. rdf:type	18
3.2.2. rdf:Property	19
3.3. RDF Wrap Up	20
4. RDFS	21
4.1. Extending RDF with RDF Schema	21
4.2. Modeling with RDFS	22
5. RDFS++ and OWL	23
5.1. Properties Supported In RDFS++	23
5.1.1. rdfs:subClassOf	24
5.1.2. rdfs:range	24

5.1.3.	rdfs:domain	24
5.1.4.	rdfs:subPropertyOf	24
5.1.5.	owl:sameAs	24
5.1.6.	owl:inverseOf	24
5.1.7.	owl:TransitiveProperty	24
5.2.	The SPARQL Query Language	24
II.	AllegroGraph Extended Tutorial	31
6.	AllegroGraph’s Extensions to RDF	33
6.1.	Examples Use Cases	33
7.	SPARQL Queries	35
8.	AllegroGraph Reasoning System	37
9.	AllegroGraph Prolog Interface	39
III.	Common Lisp Utilities for Information Processing	41
10.	Entity Extraction from Text	43
10.1.	KnowledgeBooks.com Entity Extraction Library	43
10.2.	Entity Extraction with AllegroGraph Example	43
11.	Automatic Text Tagging	45
11.1.	KnowledgeBooks.com Text Tagging Library	45
11.2.	Text Tagging with AllegroGraph Example	45
12.	Automatically Summarizing Text	47
12.1.	KnowledgeBooks.com Automatic Summarization Library	47
12.2.	Automatic Summarization and AllegroGraph Index/Search Example	47
IV.	AllegroGraph Application Examples	49
13.	Using Graphviz to Visualize RDF Graphs	51
14.	Using Open Calais with AllegroGraph	53
14.1.	Open Calais Web Services Client	53
14.2.	Storing Entity Data in an RDF Data Store	55
14.3.	Testing the Open Calais Demo System	57
14.4.	Open Calais Wrap Up	58
15.	Exporting SPARQL Query Results	59

15.1. Exporting to OpenOffice.org Spreadsheets	59
15.2. Exporting to PostgreSQL Databases	59

V. Sample Application: Semantic Web Enabled Web Site 61

16. Requirements and Design of the Semantic Web Portal Web Application	63
---	-----------

17. Web Application Back End Implementation	65
--	-----------

18. Web Interface for the Semantic Web Portal	67
--	-----------

18.1. Introduction to AllegroServe	67
18.2. Introduction to Web Actions	67
18.3. Dojo and Javascript for the Web Interface	67
18.4. Web Application Implementation	67

Bibliography	69
---------------------	-----------

List of Figures

1.1. Example Semantic Web Application 2

List of Figures

List of Tables

List of Tables

Preface

This book was written for both professional Common Lisp developers and home hobbyists who already know how to program in Common Lisp and who want to learn practical Semantic Web programming techniques using the AllegroGraph libraries from Franz. Inc.

TBD

Acknowledgements

People who contributed technical ideas for this book

TBD

People who contributed technical edits for this book

TBD

People who contributed copy edits to the material for this book

Carol Watson

1. Introduction

1.1. Who is this book written for?

There are many books on the Semantic Web and good tutorials and software on the web. However, there is not a single reference for Common Lisp developers who want to use AllegroGraph for development using technologies like RDF/RDFS/OWL modeling, descriptive logic reasoners, and the SPARQL query language.

If you own a Franz Lisp and AllegroGraph development license, then you are set to go, as far as using this book. If not, you need to download and install a free non-commercial use licensed copy at:

http://www.franz.com/downloads/clp/agle_survey

Franz Inc. has provided support for my writing this book in the form of technical reviews and my understanding is that even though you will need to periodically refresh your free non-commercial license, there is no inherent time limit for non-commercial use.

1.2. Why is a PDF copy of this book available free on the web?

As an author I want to both earn a living writing and have many people read and enjoy my books. By offering for sale the print version of this book I can earn some money for my efforts and also allow readers who can not afford to buy many books or may only be interested in a few chapters of this book to read it from my web site.

Please note that I do not give permission to post the PDF version of this book on other people's web sites: I consider this to be at least indirectly commercial exploitation in violation the Creative Commons License that I have chosen for this book.

Typical Semantic Web Application

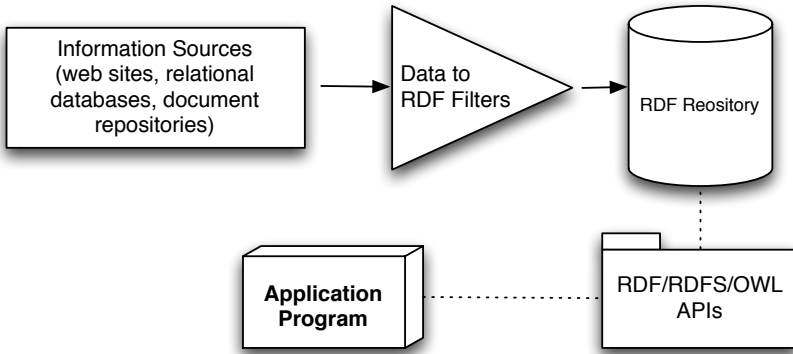


Figure 1.1.: Example Semantic Web Application

1.3. Book Software

You can download a large ZIP file containing all code and test data used in this book from the URL:

http://markwatson.com/opencontent/ag_semantic_web_code.zip

TBD: This file may not be available until January 2009

All the example code is covered by the KnowledgeBooks.com non-commercial use license for free non-commercial use. If you need to use the book software in a commercial context, a KnowledgeBooks.com commercial use license for all software examples costs \$50 per individual developer with no extra cost for deployment. This fee can be paid via a PayPal link on the <http://markwatson.com/products> web page.

The book examples are organized by chapters and each subdirectory combines the chapter number with some descriptive text.

```
chapter_02_quick_start  
chapter_06_rdf  
chapter_07_sparql  
chapter_08_reasoning  
chapter_09_prolog  
chapter_10_entity_extraction  
chapter_11_auto_tagging
```

chapter_12_auto_summarizing
chapter_13_graphviz
chapter_14_freebase
chapter_15_sparql_export
chapter_17_web_app_back_end
chapter_18_web_app_front_end
appendix_A_lisp_utilities

1.4. Why Graph Data Representations are Better than the Relational Database Model for Dealing with Rapidly Changing Data Requirements

When people are first introduced to Semantic Web technologies their first reaction is often something like, “I can just do that with a database.” The relational database model is an efficient way to express and work with slowly changing data models. There are some clever tools for dealing with data change requirements in the database world (ActiveRecord and migrations being a good example) but it is awkward to have end users and even developers tagging on new data attributes to relational database tables.

A major theme in this book is convincing you that modeling data with RDF and RDFS facilitates freely extending data models and also allows fairly easy integration of data from different sources using different schemas without explicitly converting data from one schema to another for reuse.

1.5. Book Summary

TBD

1. Introduction

2. AllegroGraph 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 AllegroGraph.
2. Theory (with some AllegroGraph short examples).
3. Detailed treatment of AllegroGraph APIs.
4. Development of Useful Common Lisp libraries information processing, data visualization, and importing 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 5 if you understand the basics of AllegroGraph. After a detailed look theory we will dig deeper into AllegroGraph development techniques in Chapters 6, 7, 8, and 9.

2.1. Starting AllegroGraph

The code snippets used in this chapter are all contained in the source file **chapter:02_quick_start/quickstart.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 the 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` t 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 here:

```
myMacBook:acl81_express markw$ ./mlisp
```

2. AllegroGraph Quick Start

International Allegro CL Free Express Edition
8.1 [Mac OS X (Intel)] (Nov 18, 2008 10:59)
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 3.0.1 [built on July 07, 2008]  
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.

2.2. 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 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).

2.2.1. 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:\temp`. 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 work with multiple repositories (and this is well documented in Franz's online documentation) for all of the tutorials, examples, and sample applications in this book we need just a single open repository.

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.

2.2.2. 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:

2. AllegroGraph Quick Start

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

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

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

2.2.3. Adding Triples

A triple consists of a subject, predicate, and object. We refer to these three values as symbols :s, :p, and :o. 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. 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*
```

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

```
<12931 containsPerson Barack Obama>
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**.

2. AllegroGraph Quick Start

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.

2.2.4. 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 NTriples format:

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

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 NTriple 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 NTriple format that we use here, the XML serialization format, or the newer and more compact N3 format.

2.3. 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 section of this book is a largely AllegroGraph independent introduction to Semantic Web technologies. If you take a while to interactively experiment with AllegroGraph before continuing to read this book then the discussion in the next three chapters may seem more “grounded” for you.

2. AllegroGraph Quick Start

Part I.

Semantic Web Technologies

3. RDF

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

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 was directly caused by trying to think about RDF in XML form. A waste of time, really, when either N-Triple or even better, N3 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:

3. RDF

- 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." 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. We will see shortly that we can define abbreviation prefixes for namespaces which makes RDF and RDFS files shorter and easier to read.

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. 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, 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" ,
```

3. RDF

```
"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 2 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 2, 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 programatically; the equivalent definition in N-Triple format 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 6. 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. 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:

4. RDFS

- 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 all of these features of RDFS when we start using the Sesame libraries in the next Chapter in Section 5.2 to perform SPARQL queries.

TBD: more detail

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 5, it 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.

Modeling and reasoning (inferencing) are tightly coupled and we will wait until our discussion of RDFS++ in Chapter 5 to experiment with the AllegroGraph reasoning and querying system.

TBD: more detail

5. RDFS++ and OWL

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

TBD: describe in detail...

DL strikes a good balance between expressiveness and computability. 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.

5.1. Properties Supported In RDFS++

There is an *unofficial* version of RDFS/OWL called RDFS++ that 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`. TBD

5.1.1. rdfs:subClassOf

TBD

5.1.2. rdfs:range

TBD

5.1.3. rdfs:domain

TBD

5.1.4. rdfs:subPropertyOf

TBD

5.1.5. owl:sameAs

TBD

5.1.6. owl:inverseOf

TBD

5.1.7. owl:TransitiveProperty

TBD

5.2. 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 (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.

We will use the N3 format RDF file `test_data/news.n3` for examples. This file was 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. We will see techniques for extracting named entities from text in Chapter 14 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 as input from another source.

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" .

```

In the following examples, we will look at queries but not the results. Please be patient: these same queries are used in the embedded Java examples in the next section 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 .
}

```

5. RDFS++ and OWL

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

TBD

Part II.

**AllegroGraph Extended
Tutorial**

6. AllegroGraph's Extensions to RDF

We saw in Chapter 3 that RDF triples contain three values: *subject*, *predicate*, and *object*. 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.

6.1. Examples Use Cases

In the following example we will extend the example started in Chapter 2 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 stoe 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 in Chapter 2 we specify values for two optional parameters to function **add-triple**:

6. AllegroGraph's Extensions to RDF

```
(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 !"r
(add-triple *demo-article* !kb:containsPerson !"Barack Obama" :c
(add-triple *demo-article* !kb:processed !"yes" :db *db* :g !"wo
```

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" .
```

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 practices 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>
```

We have seen how to perform simple triple queries/lookups in this chapter and in 2. In Chapter 7 we will see how to use the AllegroGraph APIs to perform SPARQL queries like the examples in Section 5.2.

7. SPARQL Queries

TBD

7. SPARQL Queries

8. AllegroGraph Reasoning System

TBD

9. AllegroGraph Prolog Interface

TBD

Part III.

Common Lisp Utilities for Information Processing

10. Entity Extraction from Text

TBD

10.1. KnowledgeBooks.com Entity Extraction Library

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

10.2. Entity Extraction with AllegroGraph Example

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

10. Entity Extraction from Text

11. Automatic Text Tagging

TBD

11.1. KnowledgeBooks.com Text Tagging Library

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

11.2. Text Tagging with AllegroGraph Example

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

11. Automatic Text Tagging

12. Automatically Summarizing Text

TBD

12.1. KnowledgeBooks.com Automatic Summarization Library

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

12.2. Automatic Summarization and AllegroGraph Index/Search Example

TBD

12. *Automatically Summarizing Text*

Part IV.

AllegroGraph Application Examples

13. Using Graphviz to Visualize RDF Graphs

TBD

13. Using Graphviz to Visualize RDF Graphs

14. Using Open Calais with AllegroGraph

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:

- chapter_14_opencalais/
- chapter_14_opencalais/load.lisp – loads and runs the demo
- chapter_14_opencalais/opencalais-lib.lisp – performs web service calls to find named entities in text
- chapter_14_opencalais/opencalais-data-store.lisp – maintains an RDF data store for named entities
- chapter_14_opencalais/test-opencalais.lisp – demo test program

14.1. 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 indentifies. There is a possibil-

14. Using Open Calais with AllegroGraph

ity 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 `OPEN_CALAIS_KEY` environment variable:

```
(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.asetive.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.asetive.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
              (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 airplane 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")
```

14.2. 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

14. Using Open Calais with AllegroGraph

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.

14.3. 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)
```

14. Using Open Calais with AllegroGraph

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.

14.4. 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 16, 17, and 18.

15. Exporting SPARQL Query Results

TBD

15.1. Exporting to OpenOffice.org Spreadsheets

TBD

15.2. Exporting to PostgreSQL Databases

TBD

15. *Exporting SPARQL Query Results*

Part V.

Sample Application: Semantic Web Enabled Web Site

16. 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

17. Web Application Back End Implementation

TBD

18. Web Interface for the Semantic Web Portal

TBD

18.1. Introduction to AllegroServe

TBD

18.2. Introduction to Web Actions

TBD

18.3. Dojo and Javascript for the Web Interface

TBD

18.4. Web Application Implementation

TBD

Bibliography