P3

Change Log:

10/13/2019: Provided new reference to code snippets in the Implementation section.

Learning Objectives

- Becoming familiar with an ontology editor
- Implementing a simple ontology
- Reasoning using the ontology
- Implementing a REST API to allow access to the ontology

Overview

This project consists of two parts:

- Using an ontology editor to implement an ontology. You will first get familiar with the ontology editor using a tutorial ontology. Then you will implement a new ontology described here, to be used further in the second part.
- Using Play framework to load and modify the ontology at runtime using REST API. The API will also allow the user to query information inferred by a reasoner.

Ontology Editor: Protege

Installation

Protege is a popular GUI ontology editor. You can use any editor you choose, as long as it can export the ontology in the OWL/XML format, but we highly recommend Protege.
Download the latest stable version (5.5.0) here:
https://protegewiki.stanford.edu/wiki/Protege_Desktop_Old_Versions

For Linux, you need to unzip and run the run.sh file. Steps may be slightly different for other platforms.

Tutorial

Follow the 10-minute tutorial on using Protege to implement a Pizza hierarchy (this is just to learn; not the one you'll be graded on):

https://protegewiki.stanford.edu/wiki/Protege4GettingStarted // About the UI

Follow the Pizza tutorial steps. The tutorial is old and several things have changed, so please go over the guidelines/exceptions given here before starting. Also expect the tutorial to take more than 10 minutes.
Follow the tutorial linked below, **except the following**
https://protegewiki.stanford.edu/wiki/Protege4Pizzas10Minutes // Pizza tutorial:

- Use org.coode.annotate_2_1_1.zip version for the Annotate plugin.
- The matrix plugin link is outdated in the tutorial. Use version 4.0.1 from here: https://github.com/co-ode-owl-plugins/matrix/releases
- In section “Setup tabs you will need (and not need)” and later: the “Matrix” tab is actually “Class matrix” tab.
- In section “Setup the renderer and how new entities will be created”: Do **NOT** check “auto ID” as the tutorial suggests. We will want to specify IDs in the later part of the project.
- In section “Setup the entities tab”: The “View” menu is actually “Window -> View”

In case of any issues, post on Piazza.

Hints

- “Restrictions” can be added as follows:
    - First, create your class that you want to define using a restriction (entities tab -> classes subtab, right click a class and create subclass)
    - Click on the newly added class. In the Description panel on the right, click on the plus next to EquivalentTo, then add your conditions (“some” is “someValuesFrom”, “only” is “allValuesFrom”, etc.)
    - While creating restrictions, you can directly type in a condition by going to the “class expression editor” tab after clicking the plus. For example, you could type `Pizza and ((hasBase some DeepPanBase) or (hasTopping some CheeseTopping))`
        - This would represent a restriction which has individuals that belong to Pizza, and have at least one deep pan base and at least one cheese topping.

Our Ontology

Schema

Now we shall implement our own Ontology using Protege. Here are the assertions we want to capture:

1. A Person is either a Merchant or a Consumer

2. A Transaction has one sender and one receiver (both Persons), both of which participate in the transaction
3. A Commercial transaction is one where at least one of the sender and receiver is a Merchant
4. A Personal transaction is one where both the sender and the receiver are Consumers
5. A Purchase transaction is a Commercial transaction where the receiver is a Merchant and the sender is a Consumer
6. A Refund transaction is a Commercial transaction where the sender is a Merchant and the receiver is a Consumer
7. A Merchant is Trusted if it participates in at least one Purchase transaction

Hints

These steps are not meant to be complete, but general guidelines:

- Create a new ontology, and specify some url under Active Ontology -> Ontology IRI (maybe change the last segment to something like csc750 or p3). **We will need this IRI later.**
- Start by creating a skeleton hierarchy
    - Go to entities -> classes
    - Click owl:Thing
    - Click Tools -> Create class hierarchy, and enter all the classes (tabs denote subclasses) as in the tutorial (or create one subclass at a time by right-clicking the parent class)
- Then create all the properties
    - Go to entities -> object properties
    - Make sure to mark properties as “functional” or “inverse functional” as required. For example, hasSender is functional since a transaction has a single sender.
    - Make sure to specify their ranges and domains
- Now create all the restrictions based on properties, by going back to classes

Export the ontology

File -> Save as -> OWL/XML Syntax

Individuals

You will notice that we have the schema (classes and properties) but no individuals (instances). These will be added at runtime through a REST API.

Ontology server

REST API

Our REST API will allow adding individuals (instances) and retrieving some inferred information about them. Let’s first look at the API, then we’ll figure out how to implement it.

Request	Response	Description
POST /addmerchant/:uniqueID	{ “status”: “success”}	Add an individual to Merchant class. Assume ID will be unique across *everything*, not just merchants. **Use the IDs as IRI suffixes. Example, http://your.IRI.here#merchant1**
POST /addconsumer/:uniqueID	{ “status”: “success”}	Add an individual to Consumer class. Use ID as above

POST
/addtransaction/:senderID/:receiverID/:transactionID
	{
	“status”: “success”
	}
		Add an individual to Transaction class. Use ID as above.
		
		Remember to set associated properties!

GET
/iscommercial/:transactionID
	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}
		Return whether a transaction is commercial.
		
		The transaction ID is what was supplied while creating the transaction. If you used it as an IRI suffix, you shouldn’t have any problems fetching it.

GET
/ispersonal/:transactionID
	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
		Return whether a transaction is personal.

	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}	

GET /ispurchase/:transactionID	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}
		Return whether a transaction is a purchase transaction.

GET /isrefund/:transactionID	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}
		Return whether a transaction is a refund transaction.

GET /istrusted/:merchantID	{
	“status”: “success”,
	“result”: “[true|false]”
	}
	or
	{
	“status”: “failure”,
	“reason”: “not a transaction”
	}
		Returns whether a merchant is trusted. If the ID doesn’t belong to a merchant, returns an error message.

POST /reset	{
	“result”: “success”
	}
		This should reload the ontology (or delete all added individuals), so that we can start the testing afresh.

You will notice that when creating a transaction, we don’t specify whether it belongs to commercial, etc. classes. Same with a merchant being trusted. Since these classes are implemented as restrictions, the system should automatically figure out which is which. That’s exactly the job of a **reasoner**.

Implementation

We will now create a server using Play that will use our ontology. By now, you are already familiar with how to set up a server and create REST APIs, so we will omit those details (refer to P1 and P2 if needed).

We will use **Apache Jena** with the **Openllet reasoner** to process our ontology. To use them, add the following to your build.sbt:

```
libraryDependencies += "com.github.galigator.openllet" %
"openllet-jena" % "2.6.4"
```

(Note: Openllet version 2.6.5 was released just a few days ago and has not been tested)

Apache Jena: https://jena.apache.org/documentation/ontology/
Openllet Reasoner: https://github.com/Galigator/openllet
Openllet with Jena example (follow usageWithOntModel):
https://github.com/Galigator/openllet/blob/integration/examples/src/main/java/openllet/examples/JenaReasoner.java

In case the Jena documentation is overwhelming, we have provided some sample code snippets to help you using the Pizza example. These code snippets should help you focus on the parts that are needed for this project.
https://gist.github.ncsu.edu/hpjoshi/32f849022c3a6ed7e8e38fce347b3e71

These are the relevant code snippets you should refer to and modify for your purposes. Note that even though they are separated in multiple files here you may need them in a single class file.

Imports.java: These are most of the imports you will need.
LoadOntology.java: How to load the ontology from a file.
CreateIndividual.java: Creating an individual and checking inferred classes.

Deliverables & Grading Rubric

You must submit your project as a single zip file. The grading rubric is subject to change but is given here as a guideline.

Part 0: Valid Submission (5%)

A valid submission means your submitted project executes out of box. In other words, it can be executed using ‘sbt run’ on a clean computer with only JDK and sbt installed. Make sure your project:

- Is submitted completely
- Has no compiling error
- Includes essential libraries, either in your code repository or configured as remote libraries in build.sbt script
- Includes no absolute path

A safe way is to test your program on another computer before you submit.

Remove target and intermediate files. For a sbt project, this can be achieved by deleting nested target directories, such as ./target, ./project/target, and ./project/project/target. Refer to: https://stackoverflow.com/questions/4483230/an-easy-way-to-get-rid-of-everything-generated-by-sbt

Part 1: Ontology schema (.owl file; 20%)

Make your .owl file easy to locate. Either in the top level of the folder, or inside an owl/ folder.

- Contains all the classes: 5
- Contains all the properties: 5
- Correctly identifies functional and inverse functional properties: 5
- Correctly implements restrictions: 5

Part 2: Ontology server (75%)

- Implements REST APIs for creating individuals: 30
- Implements REST APIs for inferred classes: 40
- Implements Reset API correctly: 5