Project #2: OWL Constraint Checker

Due: Tuesday, November 11

This is an individual assignment and will count for 20% of your overall grade. Note, this program is more complicated than it first appears, so be sure to start early, particularly if you are not comfortable programming in Java.

When creating OWL data, it is sometimes useful to know how well this data conforms to a given ontology. Many data sets are designed to be locally complete: that is, any relevant information about the individuals is provided within the file. It is also common to assume that individuals with different names in a file are supposed to denote different things. If we make the unique names assumption and the closed world assumption (of course, both of the assumptions are incorrect with respect to OWL in general), it is possible to create a constraint checker that determines how well each individual conforms to the structure implied by the ontology. In this assignment, you will write a tool that takes as input a simple OWL ontology and an OWL data document, and does some basic checks to see if the data document is complete with respect to the constraints of the ontology. In order to keep this task manageable, we will only be concerned with checking the following constraints:

- That the rdf:type of every instance in the data file is a class defined in the ontology file.
- If the rdf:type of an instance is a class that is defined as the subclass of an OWL DL Restriction, then the instance's properties obey the specified constraint.

Furthermore, the input ontologies we use will **not** have:

- rdfs:subPropertyOf, rdfs:domain, or rdfs:range
- RDF-style datatyping (a la the rdf:datatype attribute).
- versioning or imports information
- property characteristics, such as owl:FunctionalProperty, owl:inverseOf, etc.
- ontology mapping properties, such as owl:sameClassAs, owl:equivalentTo, etc.
- anything other than a named class as the subject of an rdfs:subClassOf statement
- multiple restrictions of the same form on the same property (e.g., two different someValuesFrom restrictions on the same property *P*).
- complex classes, such as those defined by owl:intersectionOf, owl:oneOf, etc. Note, as a consequence of this, all class descriptions that appear in an allValuesFrom or someValuesFrom restrictions will be atomic.

When checking constraints, the tool should make both the unique names assumption and the closed world assumption. That is, you can assume that if two resources have different URIs then they are distinct, and you can assume that all relevant data is specified in the data file. Therefore, if there was a minCardinality of 2 on a property, and an instance only had one value for that property, that would be considered a constraint violation that the tool should report. A particular class may have multiple constraints, and each should be checked for any instance of the class.

Your file should be run as:

`java Checker` *ontfile datafile*

where *ontfile* is the pathname of an OWL ontology file, and *datafile* is the pathname of a file with OWL instance data.

The output of your program should be:

```
No constraints violated.
```

or a list of specific error messages. In the case of an instance that is of an undefined type, report an error message of the form:

`ERROR: Undefined Type - Instance` *instance_id* `member of class` *class_id*

In the case of the violation of a restriction, the message should report the ID of the instance in which the error occurs, the type of constraint that is violated, the class in which this constraint is specified, and the property on which the constraint is placed. For example:

```
ERROR: Property Restriction Violation
       Instance: http://somewhere.org/data#band1
       Class: http://somewhere.org/ont#Band
       Property: http://somewhere.org/ont#hasMember
       Constraint: minCardinality = 2
```

In these error messages, the Class should be the class on which the property restriction is defined. Note, this may be a superclass of the instance's actual rdf:type.

Use of Jena

Every OWL file can be viewed as a set of RDF triples, thus you can use an RDF parser to read OWL. For this project, you must use Jena v. 2.5.6 to parse the input files. In order to do this the **jena.jar**, **log4j-1.2.12.jar, commons-logging-1.1.1.jar, xml-apis.jar, xercesImpl.jar, icu4j_3_4.jar** and **iri.jar** files must be in your classpath. You will need to use classes from the **com.hp.hpl.jena.rdf.model** package, and may find the **com.hp.hpl.jena.vocabulary.*** packages useful as well. Do not directly use any other packages from the Jena distribution without my permission. In particular, you are forbidden to use Jena's ontology or reasoner components.

Design Hints

Even with the simplifications specified above, this task can be challenging, so here is a suggested approach to solving this problem:

1. Create a set of Java class that can parse an ontology and store basic OWL class information, including superclasses and property restrictions. In order to help you out, I have provided three unfinished classes that you may use (**OwlOnt.java**, **OwlClass.java**, and **OwlProperty.java**). These classes provide basic data structures and some simple

access methods. However you will have to implement the methods for parsing an ontology from an OWL file, and eventually for testing if one OWL class is a subclass of another class (see Step 4 for the latter). You may modify these Java classes in any way you wish, and may also choose not to use them at all. In any case, be careful when trying to determine the superclasses of an OWL class. The property rdfs:subClassOf can be used either with a named class or with an anonymous owl:Restriction. These two forms should be treated differently by your application.

2. Write code that reads a data file, determines what the instances are, and verifies that all types correspond to a Class in the ontology file. Print an error message for each instance that fails this test. You can ignore instances that are untyped.
3. Since the cardinality and hasValue constraints should be the easiest to check, you should next write code that takes those instances that pass step 2, checks these instances for cardinality and hasValue constraints, and reports any violations (e.g., an instance has a property that doesn't have enough values, or has a property that doesn't include the value specified by a owl:hasValue restriction). Note this must be done after steps 1 and 2, because it requires the program to know the type of the instance and what constraints are applicable for that type (as specified in the ontology). Since the constraints of any superclasses of the type should also apply, be sure your program checks these as well. Note, you may ignore triples that do not use properties defined in the ontology.
4. Write code to test if one class is a subClassOf another, whether implicitly or explicitly. Because we are restricting ourselves to very simple ontologies, we only need to look at the explicit rdfs:subClassOf relations and any transitive inferences that result (e.g., if A is a subclass of B, and B is a subclass of C, then we can conclude that A is also a subclass of C). I emphasize that no description logic inference is needed.
5. Write code to check the owl:allValuesFrom and owl:someValuesFrom constraints. Note that you must take extra care when checking these kinds of constraints. An instance of a class is also an instance of all of its superclasses, so be sure to take this into account when you try to determine if the value of a property is of the type specified by the constraint.

Submission Instructions

This assignment is due by the beginning of class on Tuesday, November 11. Create a zip or tar file that contains both your source code (.java) and compiled (.class) files (but do not include any of the Jena files in it). If you used the three files I provided, make sure they are included. Also, if you use a .BAT file or some other form of script to compile and run your program, please include this as well. The electronic version of your program file must be submitted using the course webpage on the Blackboard Learning System (see https://ci.lehigh.edu/). Select Assignments, and then click on "View/Complete" for Project #2. You will then be able to attach your file for submission. Once you are absolutely certain that your file is complete, press the "Submit" button (although you can add your file at any time, I will not be able to access it until you press "Submit"). Also print out your .java files, and turn them in during class. All of your files should be reasonably commented, including an initial comment that identifies you as the author and descriptive comments for each class and method.