COMP2212 Programming Language Concepts Coursework

Dr Julian Rathke

Semester 2 2021/22

Introduction

In this coursework you are required to *design and implement* a domain specific programming language for querying simple RDF documents. There are many query languages for assorted data formats, the most famous perhaps being SQL for relational databases. You are welcome to research any existing query languages to inspire the design of your own language. If you want, you could stick closely to the syntax of an existing language. Of course, you are also more than welcome to go your own way and be as original and creative as you want: it is *YOUR* programming language, and your design decisions.

You are required to (1) invent an appropriate syntax and (2) write an interpreter (possibly using `Alex` and `Happy` for lexing and parsing). Your overall goal is :

> To design and implement a programming language for querying and manipulating simple RDF input files.

For the five example problems listed below, you will be required to produce a program, in your language, that solves each of the problems. These five programs, together with the Haskell sources for your interpreter, are the required deliverables for the first submission.

Please keep a record of all the sources you consult that influence your design, and include them in your programming language manual. The manual will be required for the second submission, along with programs for five additional unseen problems, which we will make public *after* the deadline for the first submission. You should anticipate that the additional problems will comprise of variations and combinations of the first five problems. You will find more procedural details at the end of this document.

The specification is deliberately loose. If we haven't specified something, it is a design decision for you to make: e.g. how to handle syntax errors, illegal inputs, whether to allow comments, support for syntax highlighting, compile time warnings, any type systems etc. A significant part (50%) of the mark will be awarded for these qualitative aspects, where we will also take into account the design of the syntax and reward particularly creative and clean solutions with additional marks. The remaining 50% of the mark will be awarded for correctly solving a number of problems using your programming language. The correctness of your solution will be checked with a number of automated tests. We will release an "automarker" script before the first deadline which will give you an indication as to how your programs will perform in our test harness.

The Input and Output Format

For each problem we will declare the name of one or more input files in a simple RDF format. The particular variant of RDF we will be using is Turtle (see https://www.w3.org/TR/turtle/) with some restrictions. The input file name will correspond to a file in the current working directory with an extension ".ttl". For example, if the problem input file is called "foo" and your interpreter is executed in directory "C:/Home/Users/jr/STQL/" then the input file will be found at "C:/Home/Users/jr/STQL/foo.ttl".

The input files will be in a simple Turtle format as follows. They may contain a base URI annotation using `@base <URI> .` and there may be a number of prefix declarations `@prefix name : <URI> .` present. These are used as a convenience to make the data triples more human readable. Otherwise the input files contain a number of triples: each triple is of the form

```
  <Subject> <Predicate> <Object> .
```

where `Subject` and `Predicate` are both URIs (possible defined in shorthand using the base URI or a named prefix), but `Object` is either a URI or a literal value. You may assume that URIs contain no special characters and are always of the form

```
     http://dom1.dom2.  ...  .domN/subDomain1/subDomain2/.../subDomainM/#tag
```

where the sub-domains and `#tag` are optional and there is at least 1 top level domain. For example we may have

```
@base <http://www.example.org/> .
@prefix foo : <http//www.exampleprefix.org/> .
@prefix bar : <exampleBasePrefix/> .
  <http://www.exampleabsolute.org/exSub1>
    <http://www.exampleabsolute.org/exPred1>
      <http://www.exampleabsolute.org/obj1> .
  <exBaseSub> foo:exPred2 <exObj2> .
  bar:exSub2  <exPred3> "LiteralValue" .
```

and this is equivalent to the absolute triples

```
  <http://www.exampleabsolute.org/exSub1>
    <http://www.exampleabsolute.org/exPred1>
      <http://www.exampleabsolute.org/obj1> .
  <http://www.example.org/exBaseSub>
    <http//www.exampleprefix.org/exPred2>
      <http://www.example.org/exObj2> .
  <http://www.example.org/exampleBasePrefix/exSub2>
    <http://www.example.org/exPred3>
      "LiteralValue" .
```

The literal values range over String values containing only alphabetic characters of upper and lower case, Integer values that match the regexp "[+-]?[0-9]+" and Boolean values `true` and `false`.

In addition to this there are two shorthand forms for describing triples. Where the triples share a subject we may write a *Predicate List* as

```
<Subject> <Predicate1> <Object1> ;
          <Predicate2> <Object2> ;
          ...
          <PredicateN> <ObjectN> .
```

and where the triples share both a subject and predicate we may write an *Object List* as

```
<Subject> <Predicate> <Object1> , <Object2> , ... <ObjectN> .
```

We will not be using empty prefixes, blank nodes or collections. You may assume in the stated problems that all input files will be well-formed in this simple Turtle format.

For each stated problem, the output should be in the same simple Turtle format where all URIs are expanded to absolute addresses (no base or prefix annotations) and there is no use of Predicate or Object Lists. That is, it is a file of pure triples. Moreover, the output should always be **sorted lexicographically** by subject, predicate, object, with the default Haskell value ordering for String, Integer and Boolean types. **The output should always be printed to Standard Out**.

Problems

For every input name `foo` used in a formula, you may assume that we will place a Turtle file called `foo.ttl` in the same directory where we execute your interpreter. The file will always be compatible with the Turtle format. You may assume that we will not require you to perform any additional operations on the IRIs or literal values other than those indicated by the problems given below.

Problem 1 - Conjunction

Assume two input files `foo` and `bar`. Output a single combined Turtle file that contains all of the triples of both `foo` and `bar`. Be careful about name clashes between the two input files.

Problem 2 - Simple Pattern Matching

Assume one input file `foo`. The output should contain all triples `<Subj>,<Pred>,<Obj>` of `foo` where `Subj` is `http://www.cw.org/#problem2` and `Obj` is the boolean literal `true`.

Problem 3 - Further Pattern Matching

Assume one input file `foo`. The output should contain all triples `<Subj>,<Pred>,<Obj>` of `foo` where `Pred` is one of the following:

`http://www.cw.org/problem3/#predicate1`
or
`http://www.cw.org/problem3/#predicate2`
or
`http://www.cw.org/problem3/#predicate3`

Problem 4 - Linking

Assume two input files `foo` and `bar`. Output a single combined Turtle file that contains all of the triples `<Subj>,<Pred>,<Obj>` of `foo` where `Obj` appears as a subject of a triple in `bar` and all of the triples of `<Subj>,<Pred>,<Obj>` of `bar` where `Obj` appears as a subject of a triple in `foo`.

Problem 5 - Graph Edits

Assume one input file `foo` that contains a number of triples all of whose objects are Integer literals. For each such triple `<Subj> <Pred> n` in `foo` where `n` is less than 0 or greater than 99, the output should contain a triple of the form `<Subj> <http://www.cw.org/problem5/#inRange> false .` For each such triple `<Subj> <Pred> n` in `foo` where `n` is between 0 and 99 inclusive, the output should contain the triple

`<Subj> <Pred> n+1` as well as the triple
`<Subj> <http://www.cw.org/problem5/#inRange> true .`

First submission - due Thursday April 28th 4pm

You will be required to submit a zip file containing:

- the sources for the interpreter for your language, written in Haskell
- five programs, written in **YOUR** language, that solve the five problems specified above. The programs should be in files named `pr1.stql`, `pr2.stql`, `pr3.stql`, `pr4.stql`, `pr5.stql`.

We will compile your interpreter using the command `ghc Stql.hs` so you will need to include a file in your zip file called `Stql.hs` that contains a main function along with any other Haskell source files required for compilation. Prior to submission, you are required to make sure that your interpreter compiles **on a Unix machine with a standard installation of GHC (Version 8.4.3) or earlier**: if your code does not compile then you will be awarded 0 marks. Before submission, we will release a simple "automarking" script that will allow you to check if your code compiles and whether each of your programs passes a basic test.

You can use Alex and Happy for lexing and parsing but make sure that you include the generated Haskell source files obtained by running the `alex` and `happy` commands as well as the Alex and Happy source files. Alternatively you can use any other Haskell compiler construction tools such as parser combinator libraries. You are welcome to use any other Haskell libraries, as long as this is clearly acknowledged and the external code is bundled with your submission, so that it can compile on a vanilla Haskell installation.

Interpreter spec. Your interpreter is to take a file name (the program in your language) as a single command line argument. The interpreter should produce output on standard output (`stdout`) and error messages on standard error (`stderr`). For each problem, we will test whether your code performs correctly by using a number of tests. We only care about correctness and performance will not be assessed (within reason - our marking scripts will timeout after a generous period of time). You can assume that for the tests we will use correctly formatted input. For example, when assessing your solution for Problem 1 we will run

```
./stql pr1.stql
```

in a directory where we also provide our own versions of `foo.ttl` and `bar.ttl`. We will then compare the contents of `stdout` against our expected outputs. Whitespace and formatting is unimportant as long as the output is sorted, uses pure Turtle triples and contains no other text.

Second submission - due Friday May 6th 4pm

Shortly after the first deadline we will release a further five problems. Although they will be different from Problems 1-5, you can assume that they will be closely related, and follow the same input/output conventions. You will be required to submit two separate files.

First, you will need to submit a zip file containing programs (`pr6.stql`, `pr7.stql`, `pr8.stql`, `pr9.stql`, `pr10.stql`) written in your language that solve the additional problems. We will run our tests on your solutions and award marks for solving the additional problems correctly.

Second, you will be required to submit a 5 page report on your language **in pdf format** that explains the main language features, its syntax, including any scoping and lexical rules as well as additional features such as syntax sugar for programmer convenience, type checking, informative error messages, etc. In addition, the report should explain the execution model for the interpreter, e.g. what the states of the runtime are comprised of and how they are transformed during execution. Languages that support strong static typing and type safety with a formal specification are preferred. This report, together with the five programs will be evaluated qualitatively and your marks will be awarded for the elegance and flexibility of your solution and the clarity of the report.

Please note: **there is only a short period between the first and second submission**. I strongly advise preparing the report well in advance throughout the development of the coursework.

As you know, the coursework is to be done in groups of three. Only one submission per group is required. I don't need to know who is in which group at the first submission but as as part of the **second** submission we will require a declaration of who is in your group and how marks are to be distributed amongst the members of your group. You will receive all feedback and your marks by Friday May 27th. Please ensure that it is the same group member that submits for both the first and second submissions.

Marks. This coursework counts for 40% of the total assessment for COMP2212. There are a total of 40 marks available. These are distributed between the two submissions as follows:

You will receive the test results of Submission One prior to the second deadline but no marks will be awarded until after Submission Two.

After Submission Two we will award up to 20 marks for the qualitative aspects of your solution, as described in your programming language report. We will also award up to 20 marks for your solutions to the ten problems. For each problem there will be 2 marks available for functional correctness only.

You have the option of resubmitting the interpreter after receiving your testing results from Submission One. This will however incur a 50% penalty on the marks for functional correctness of all ten problems. Therefore, if you decide to resubmit your interpreter in the second submission the maximum possible total coursework mark is capped at 30 marks (20 for the report plus 10 for functional correctness).

Any late submission to either component will be treated as a late submission overall and will be subject to the standard university penalty of 10% per working day late.