COMP2212

DSL for querying RDF turtle format documents

January 9, 2024

Ishaipiriyan Karunakularatnam
ik3g20@soton.ac.uk

Danny M Hibbert
dmh1g19@soton.ac.uk

Robert Stoica
rss1g20@soton.ac.uk

Contents

1 Introduction

This report discusses the design and implementation of a domain specific programming language for querying RDF documents in turtle format with bases and prefixes. The language is written in Haskell using Alex for tokenizing and Happy for parsing.

2 Language Design

This language stands out predominantly from other general query languages in that it only deals with turtle formats strictly containing triples of the form <subject><predicate><object>. The design of the language closely relates to standard RDF querying languages with a simplified and straight forward approach, literals are always expected to be of an integer format.

The language uses a set of keywords to filter turtle files after specifying importing instructions and accompanying variable assignments, type checking is performed through the use of the program structure - not explicit data for type scoping is defined.

3 Syntax

The syntax of this language is supposed to be a very simplified take on SQL for turtle languages that only queries/edits and returns turtle RDF formatted text. Functions are defined in capital letters and are pre-defined by the language and cannot be overloaded. File names can be assigned a user defined variable, and any filters can be provided within **WHERE** blocks.

3.1 Program semantics

The semantics of the program follows Big Step operational semantics. We parse the program bottom up, which is then reversed before evaluation each step of the program one by one while building up to the output. A relationship is maintained between each term and their associated values during run time to output a final result. As part of the interpreter we implemented a simple CEK machine through the use of an Environment, Kontinuation and Frame list to evaluate each step of the program. We used the environment to store the values and Kontinuation to make sure the program follows our defined syntax. We focused on big step operational semantics as our program heavily focuses on producing specific values as outputs.

3.2 Scope

Due to the implementation of the language structure and the method used for tokenizing each line - scope of variables are kept globally accessible before exporting the final output.

4 Functions

4.1 IMPORT

The language scopes **.tll** files using the **IMPORT** function. The function is called before retrieval of triples and any application of filters.

```
IMPORT foo.txt...
```

4.2 AS

AS provides a reference assignment **Var** for using in filtering processes within **WHERE** blocks.

```
IMPORT foo.txt AS A...
```

4.3 INTO

The `.tll` file to write to is defined at the start of the main block of code using the **INTO** function.

```
INTO out1...
```

4.4 GET

The triple format is specified as the data to retrieve from the imported **.tll** file through the use of **GET** followed by a list.

```
GET [subj,pred,obj]...
```

4.5 WHERE

The **WHERE** block is followed by a section in curly brackets containing all filters to be applied to the triple defined in the **GET** block.

```
WHERE A[subj,pred,obj]...
```

4.6 EXPORT

Outputting the final written **.tll** file can be done after the main code block of the program using the **EXPORT** function followed by the filename defined earlier in the program using the **INTO** function.

```
EXPORT out1...
```

4.7 LISTS

Lists are depicted within square brackets and can contain element definitions in regards to turtle style triples and other literal values such as bools. Lists can represent the contents of different files with suffixes.

```
WHERE [subj,pred,obj]...
WHERE B[subj,pred IN A,obj]...
WHERE A[subj IN B,pred IN C,obj IN D]...
```

4.8 Conditionals

Conditionals are used to filter through more complex conditions

5 Grammar

⟨***prog***⟩ ::= ⟨*stmt*⟩
| ⟨*prog*⟩ ⟨*stmt*⟩

⟨***stmt***⟩ ::= ⟨*exp*⟩ ;

⟨***exp***⟩ ::= ⟨*int*⟩
| ⟨*alpha*⟩
| ⟨*function*⟩
| ⟨*int*⟩ ⟨*operator*⟩ ⟨*int*⟩
| ⟨*ifStatement*⟩
| (⟨*exp*⟩)

⟨***function***⟩ ::= INTO ⟨*exp*⟩ ⟨*exp*⟩
| GET [⟨*list*⟩] WHERE ⟨*list*⟩
| IN ⟨*exp*⟩
| AS ⟨*exp*⟩
| Import ⟨*exp*⟩ AS ⟨*exp*⟩
| EXPORT ⟨*exp*⟩

⟨***operator***⟩ ::= `<`
| `>`
| `+`
| `-`
| `<=`
| `>=`

⟨***ifStatement***⟩ ::= IF ⟨*exp*⟩ THEN ⟨*exp*⟩ ELSE ⟨*exp*⟩

⟨***list***⟩ ::= ⟨*listContent*⟩
| ⟨*listContent*⟩ , ⟨*list*⟩

⟨***listContent***⟩ ::= subj
| pred
| obj
| subj IN ⟨*exp*⟩
| pred IN ⟨*exp*⟩
| obj IN ⟨*exp*⟩
| ⟨*bool*⟩
| ⟨*exp*⟩

⟨***listCompare***⟩ ::= [⟨*list*⟩]
| [⟨*list*⟩] ⟨*comparison*⟩ ⟨*listCompare*⟩

⟨***comparison***⟩ = OR
| AND

⟨***bool***⟩ ::= true
| false

⟨***int***⟩ ::= [0-9]

⟨***alpha***⟩ ::= [a-z]
| [A-Z]

6 Example code

6.1 Linking from different files

```
IMPORT foo AS A;
IMPORT bar AS B;

INTO out4 GET [subj,pred,obj] WHERE {A[subj,pred,subj IN B] AND B[subj,pred,subj IN A]};

EXPORT out4;
```

6.2 Writen output using conditionals

```
IMPORT foo AS A;

IF {A[obj < 0 OR obj > 99]} THEN
INTO out5 WRITETRUE {A[subj,http://www.cw.org/problem5/#inRange,false]}
ELSE IF {A[obj >= 0 AND obj <= 99]} THEN
INTO out5 WRITETRUE {A[subj,pred,obj + 1] AND A[subj,http://www.cw.org/problem5/#inRange,true]}
ELSE NOTHING;
EXPORT out5;
```

6.3 Pattern matching triple conditions

```
IMPORT foo AS A;

IF {A[obj < 0 OR obj > 99]} THEN
INTO out5 WRITETRUE {A[subj,http://www.cw.org/problem5/#inRange,false]}
ELSE IF {A[obj >= 0 AND obj <= 99]} THEN
INTO out5 WRITETRUE {A[subj,pred,obj + 1] AND A[subj,http://www.cw.org/problem5/#inRange,true]}
ELSE NOTHING;
EXPORT out5;
```

7 Type checking and error handling

The language is designed to be as simple as possible and perform any type checking sequentially during the interpretation. The program will catch and throw errors in regards to input that does not adhere to a passable AST.

8 Language review

The implementation of the language lacks appropriate error checking methods, flexibility and misses some crucial implementations to solve more complex problems. There are areas to be improved with the

language such as union operations as well as the inclusion of a unary minus operator to accommodate for negative numbers.