COMP2212
Our Query Language

Edward Clewer
`ejc1g20@soton.ac.uk`

Thomas Cutts
`tc3g20@soton.ac.uk`

Hristiyan Georgiev
`hhg1u20@soton.ac.uk`

1 Introduction

This report highlights the syntactical and semantic properties of our language. It follows the chronological order of our work, explaining our decision-making along the way.

2 Turtle Files

2.1 Reading files

The aim of the turtle parser is to take in a file in the Turtle RDF format and to create a list of triples in raw form which will be easier to work with. Our target output type for the finished turtle parser is a list of triples of type [(String, String, String)].

2.2 Monadic Parsing

Drawing inspiration from the Programming III coursework, we decided to employ a monadic parser using the provided parsing library, Graham Hutton's 'Parsing.hs'. Our parser which uses this parsing library is called *TTLParser.hs*. For each line that it reads, it expects either a normal triple (Subject, Predicate, Object), a Predicate List, an Object List, a base, or a prefix. It is then stored to the output as an item in a list: (Subject, Predicate, Object). Bases and prefixes are also identified as lines that the parser can recognise and stored in the output list with their annotation for future processing.

2.2.1 Error Checking

The monadic parser will fail if the file does not match the correct format of a turtle RDF file and no invalid data will be passed on to the main body of the code. Where possible, we have tried to accept as many possible spaces separating any text where the meaning of the data would not change at all.

2.3 Expanding Lists and Substituting Bases and Prefixes

The output from the monadic parser is then passed to the main body of the code which takes the correctly separated lines, bases, prefixes and lists to transform into a single list of all the triples that the input file described. We decided that it would be easiest to work with a list of triples that all have the same format, where the base and prefixes have been 'substituted' and the lists have been 'expanded'. A second file *Read.hs* is used to perform these operations on the data which has been parsed from the turtle file.

2.3.1 Classifying the Lines

Each line from the parser is given to *Read.hs* and classified as either: a triple, object list, predicate list, base or prefix. The lists are expanded and the bases and prefixes are substituted.
Once all of the triples are in this form, the 'sort' function is called on the list of triples and a list of triples of the form [(Subject, Predicate, Object)] is returned. This puts the data into a fully expanded and substituted form which means that

not only is the data easy to work with in our own programming language, but also it is in the correct format to be outputted after the operations in the challenges and another sorting operation.

3 Tokens and Grammar

3.1 Lexical Analysis

We used the lexical analyser Alex to generate tokens for our language. The operation tokens are in capital letters, following the design principles of popular querying languages such as SQL. We use an 'END' token to solve the 'dangling-else problem', with 'THEN' and 'END' defining an 'if statement's' scope.
Using the 'posn' wrapper allows us to track the line and column of the tokens in the input text, facilitating more informative error messages.

3.2 Parsing

We used the parser-generator Happy to make the parser for our language.Manipulating associativity and precedence allowed us to eliminate shift-reduce conflicts in the grammar. Additionally, we gave the multiplication and division operators higher precedence than addition and subtraction to preserve mathematical correctness.

Upon hitting a parsing error, an informative error message details the line and column, as well as the token at which parsing failed, thus improving error handling.

3.3 Syntax Highlighting

We have configured syntax-highlighting for the IDE 'Notepad++' (in the form of an XML file). We distinguish different groups of commands by highlighting comments, query operators, mathematical and logical operators, and the keywords SUBJ, PRED, and OBJ in different colours to aid readability.

4 Interpreter

4.1 Execution Model

Any program in our language must begin by reading a file. Our Grammar allows for 3 different constructors to begin the AST which the program creates, and they are our entrance to the interpreter and unwrapping the tree.

4.1.1 Reading Files

Each program begins with

```
READ fileName1 fileName2 ... fileNameN
```

The interpreter then reads the separate files, composes all the triples and attaches the context of each file. We refer to context as the file which a certain list of triples refers to. The context of the triples in **fileName1** would be **fileName1**. All triples and their contexts are in the **scope** of each **READ** operation. (See scope section below).

4.1.2 Query Conditions

A **READ** operation can then finish and reach the end of the program, in which case the triples are simply output, or it can have numerous conditions:

```
READ fn1 .. fnN WHERE Condition(s)

READ fn1 .. fnN WHERE IF Condition(s) THEN Action(s) END

READ fn1 .. fnN WHERE IF Condition(s) THEN Action(s) ELSE Action(s) END

READ fn1 .. fnN WHERE Condition(s) AND IFStatement(s)
```

The interpreter takes the list of each filename and its context, removes the list of triples that are not within the context of the condition(s), then checks the condition(s) against each triple accordingly and removes those that do not hold the query. For example:

```
READ foo bar WHERE foo.OBJ+2 >= 5*3
```

The interpreter will read **foo** and **bar**, then derive the context of the condition, which is **foo**. It will remove **bar** from the output as it is not within the context. Then the interpreter will evaluate each side of the equation for each triple, and output the ones that hold the condition.

An **IF** statement without an **ELSE** condition will apply action(s) to the triples that hold the condition, and will output the rest of the triples as they are. An **IF-ELSE** statement will apply given actions to triples that hold, and those that do not hold the condition.

An **IF** Statement can also include a **Nested IF** statement, in which case the context of evaluation will be passed to each consequent nested if.

4.1.3 Actions

Each **IF Statement** includes **Actions** that can be applied to triples:

```
IF .. THEN CHANGE SUBJ PRED OBJ END
IF .. THEN INSERT SUBJ PRED OBJ END
IF .. THEN DROP END
```

A **CHANGE** action will change the queried triple to the arguments passed to it. An **INSERT** action will insert a new triple inside the context of the triples with the arguments passed to it, which are properly type-checked (See type checking below). A **DROP** action will remove the queried triple.

We have the power to sequence as many actions as we want. They are executed sequentially:

```
... CHANGE SUBJ PRED OBJ AND
    INSERT SUBJ PRED 5 AND
    INSERT SUBJ PRED True AND
    CHANGE SUBJ PRED PRED AND
    DROP ...
```

The interpreter will first change each triple to the arguments passed to the first **CHANGE** action. It will then insert 2 triples with different objects of type Int and Bool. It will then change the triples again, and drop all of them at the end.

4.1.4 Scope

Each operand - a file, a triple or its declaratives, has a scope in which they have a meaning:

```
READ foo AND READ bar WHERE foo.OBJ ... -- invalid
```

Each file is in the scope of its **READ** operation, so "foo" and "bar" have meanings in different places of the code.

Furthermore, a certain list of triples may not have a meaning within the scope of an **IF Statement**:

```
READ foo bar WHERE IF (foo.OBJ < 5) .. actions on bar -- invalid
```

Since we query only on the context of "foo" within the **IF Statement**, an action performed on "bar" will not be recognised and such will throw an error.

4.2 Error Messages

The interpreter has a wide variety of error checking and the ability to output a correct error message. The **Tokens** and the **Grammar** are designed to output a message for every single token that is misplaced in the program, and the position of the faulty piece of code. For example:

```
READ foo END -- throws an error:
ERR: Parse error at line:column 1:10 -> Misplaced token END
```

Throughout computation, the interpreter will throw an error in case of any type of ambiguity, for example:

```
foo.OBJ + bar.OBJ + cat.OBJ = 5
ERR: Ambiguous condition for query
```

The query condition above is ambiguous since one of the sides of the equation carries more than 1 context, so it is unclear from which file need the triples be queried.

Additionally, performing actions can be described with a nice syntax sugar where we have only 1 context to perform them on:

```
READ foo WHERE IF (..) THEN INSERT SUBJ PRED OBJ END
```

We can omit the context of **foo** from **foo.SUBJ** and only express the declarative required. A thorough verbose error message will be thrown in case of performing this syntax sugar on several contexts at once - it will output the type of action that the error occurred in, as well as the number of contexts being in the scope:

```
READ foo bar WHERE IF (..) THEN INSERT foo.SUBJ PRED foo.OBJ END
ERR: Ambiguous INSERT PREDICATE query, number of contexts: 2
```

There are other smaller types of error messages that refer to more detailed queries, for example "nonINT" will be thrown when we truly need the whole list of triples' objects to be integers.

Further error messages are output by performing type checking. (See below)

4.3 Type Checking

The interpreter's type checking system makes sure that different operations are correctly performed and only so on the types that support them. Some of our type checking is declared in our Grammar, for example:

```
foo.OBJ < "notInt"
```

This query will throw an error as a misplaced **String**, because operations are defined to be valid only if both sides of any inequality are **Integers**. Alternatively:

```
foo.SUBJ < 5
```

will also throw an error, because Subjects and Predicates can only be URIs, and we can only compare them by checking equality. Furthermore, **foo.SUBJ + 3** will also be invalid as they are not both integers.

Throughout program evaluation, we might want to calculate:

```
foo.OBJ * 10
```

In this case, the interpreter does further type checking on every single triple's object, selecting only those that are integers, and applying the condition on them. This allows for flexibility where not all triples may have integers as objects, but we can still query them by type checking appropriately.

As is known, turtle triples support an **Object** as type **Integer, Bool, URI** or **Literal Value/String**. **Subject** and **Predicate** are only allowed to be **URIs**, therefore it is important to have valid types in the output, even when we have custom triples from **Actions**:

```
CHANGE "literal" "<http://www.ex.org/subj>" True
ERR: Bad CHANGE query: SUBJECT and PREDICATE declarations must be
of URI format
```

The interpreter performs type checking on the input of our **Actions - CHANGE and INSERT**, by verifying that the **Subject** and **Predicate** are URIs before moving on to the next step of evaluation.

For all types of type checking, the interpreter has integrated functions to derive whether a given input/output is an **Integer, Bool, String** or **URI**. This ensures that functions that take **Integers** and output an **Integer** stay valid, or that we can compare equality between all of them by **type casting** each one as a **String**, all of which ensures the **type safety** of the program.