COMP2212 Coursework Report

Yan Zhu
yz5m20@soton.ac.uk

Yiheng Lu
yl15u20@soton.ac.uk

Yuchen Zhao
yz4u20@soton.ac.uk

May 6, 2022

1 Introduction

The language we develop, the "STQL" language, is a query language based on the Haskell. The purpose of the language is to select specific data in RDF documents. We have investigated many other query languages mainly SQL to get the basic idea. Because of some intrinsic properties of Haskell such as pattern matching, it is easy to implement query language based on that. We decided to make full use of those properties and inherit them into our language.

2 STQL language

2.1 Grammar

A program/function is defined as Prog, which is defined as

```
Prog ::= Def| Def Prog
```

Where Def is an expression of the variable name and the expression related to it:

```
Def ::= Var = Expr
```

Variables are meta types such as :

```
Var =
Bool =
```

We added some regular used expressions to the definition of Expr such as read ttl file, read and show. Also, to distinguish from Haskell, we used some different symbols to express some actions, for example, we use ".&" to express apply action instead.

```
Expr ::= if Expr then Expr else Expr
     | let Var = Expr in Expr
     | case Expr of CaseClauses end
     | proc Var -> Expr
     | Expr + Expr
     | Expr - Expr
     | Expr * Expr
     | Expr / Expr
     | Expr : Expr
     | Expr < Expr
     | Expr && Expr
     | Expr || Expr
     | not Expr
     | []
     | '(' Expr ',' Expr ',' Expr ')'
     | readTtlFile Expr
     | Expr .$ Expr
     | Int
     | Bool
     | String
```

```
        | Var
        | '(' Expr ')'
        | print Expr
        | read Expr
        | show Expr
        | nl
        | space
```

The priority of operation is the same as in other languages, which is in increasing order of:

```
                ||, &&, <, :, + -, * /, .$
```

- The "||","&", and ":" are associated to right.
- "<" does not have any association.
- "+", "-", "*", "/" and "&" are associated to left.

The case selection in our language is similar to the ones in Haskell, using a recursion definition, with usage of pattern matching:

```
        CaseClauses = CaseClause | CaseClause CaseClauses
        CaseClauses = '|' Pattern -> Expr
```

About pattern matching, we forget to set the constant variable as one of the patterns, so our language can not match constants such as "1", "ok". But this can be bypassed by pattern match variable first, and then checking whether the variable is the desired constant:

```
    Pattern = Var
            | Pattern : Pattern
            | '(' Pattern ',' Pattern ',' Pattern ')'
            | []
```

2.2 The interpret of stql

Our language uses static allocation and call by value to simplify the work of compiler. We first define a data type "Value" to express the result of expression:

```
       data Value = VString String
          | VInt Int
          | VBool Bool
          | VList [Value]
          | VTriple Value Value Value
          | VUnit
          | VErr
          | VClosure Closure
```

Where closure is the defined as :

```
       data Closure = Closure String Expr Env
```

2.2.1 Type Env:

Where "Env" is a map associating a variable name and the value of the variable, representing the environment where closure is initialized:

```
type Env = M.Map String (IORef Value)
```

2.2.2 Type App:

Using IO as a base monad, applying "StateT Env" to it to obtain:

```
type App = StateT Env IO
```

2.2.3 Execution:

When executing a program/function, our interpreter will execute every "def" in the "Prog" one by one:

```
exec :: Prog -> App ()
exec prog = forM_ prog execDef
```

2.2.4 Assigning a variable (x = e1):

First create a reference r, pointing to the value of x. Record the pointing in the environment. Under this new environment, evaluate the value of expression e1, marking it as v1, and finally pointing r to v1:

```
execDef :: Def -> App ()
execDef (name, expr) = do
        valueRef <- liftIO $ newIORef VUnit
        modify $ M.insert name valueRef
        value <- evalExpr expr
        liftIO $ writeIORef valueRef value
```

2.2.5 Evaluating expression:

```
evalExpr :: Expr -> App Value
```

I. **Calculate e1 .* e2 under "env"**

- a) Evaluate e1 and e2 under "env"
- b) The result of e1 should be a closure (Closure arg body env')
- c) Create a reference v2_ref, pointing to v2, which is the evaluated value of e2
- d) Extend "env'", introducing the relation between "arg" and "v_ref"
- e) Evaluate the entire body under the extended environment,and save the result as v3

- f) Restore the environment to "env", return v3

```
App e1 e2     -> do
  v1 <- evalExpr e1
  case v1 of
     VClosure (Closure arg body env) -> evalExpr
     e2 >>= \v2 -> liftIO (newIORef v2) >>= \v2_ref ->
      local (\_ -> M.insert arg v2_ref env) $ evalExpr body
            _   -> pure VErr
```

II. **Evaluating let x = e1 in e2**

- a) Create a reference "ref_x"
- b) Bind "x" and "ref_x" in the environment "env", save as new environment "env'"
- c) Evaluate "e1" in new environment, and save as "v1"
- d) Pointing "ref_x" to "v1"
- e) Evaluate "e2" in new environment, and save as "v2"
- f) Restore environment to "env". Return "v2"

```
    Let name e1 e2   -> do
            valueRef <- liftIO $ newIORef VUnit
            local (M.insert name valueRef) $ do
                v1 <- evalExpr e1
                liftIO $ writeIORef valueRef v1
            evalExpr e2
```

III. **Evaluating case pattern matching**

- a) Evaluating "e1", save the result as "v1".
- b) Find a pattern match from top to bottom
- c) If there is a match, then create a reference for every variable that matches the pattern and point the reference to the variable.
- d) Extend the environment.
- e) Evaluate the body.
- f) Restore the environment.
- g) Return the result of body.

```
CaseOf e1 cases   -> do
       v1 <- evalExpr e1
   case [(bindings, caseBody) | (pattern, caseBody) <-
       cases, bindings <- patternMatch v1 pattern] of
           (bindings, caseBody) : _        -> do
           bindings' <- forM bindings $ \(name, value)
               -> (,) name <$> (liftIO $ newIORef value)
               local (M.union (M.fromList bindings'))
                 $ evalExpr caseBody
                  _       -> pure VErr
```

IV. **Evaluate readTtlFile e1**

- a) Evaluate "e1"
- b) Read the related file, and decode an rdf
- c) Torn rdf into "Value"

```
ReadTtlFile e1  -> do
   v1 <- evalExpr e1
   case v1 of
       VString fname -> liftIO (readSimpleRDF fname) >>=
\rdf -> case rdf of
           Nothing   -> pure VErr
           Just rdf' -> pure $ value_of_simpleRDF rdf'
       _             -> pure VErr
```

2.3 Decode of .ttl document:

There are some "Item"s in every .ttl file. For each "Item", it should appear in the following mode: A declaration of one prefix and one ur, or a triple:

```
@prefix: <uri> .
@base <uri> .
```

Or:

```
<sub> <pre> <obj>, <obj>, <obj>, ...,  <obj>;
   <pre> <obj>, <obj>, <obj>, ...,  <obj>;
                      ....
   <pre><obj>, <obj>, <obj>, ...,  <obj>.
```

Thus, define the type of RDF as:

```
type RDF = [Item]

data Item = Prefix Name URI
          | Triple URI [(URI, [Object])]
                       deriving Show
```

For an object, there are four possible modes:

```
data Object = URIObj URI
            | StrObj String
            | IntObj Integer
            | BoolObj Bool
```

The interpreter function fo RDF (find more in the decode.hs):

```
readRDF :: String -> IO (Maybe RDF)
```

At this point, we finished the deserialization of .ttl document. However, we can still decode more of the contents in the file since a lot of uri in the file are represented with prefix, and some of the triples are contained in the list. We should replace the prefix with a certain uri and extend the grammar sugar.

Define a new dayt type, where exclude the prefix of .ttl:

```
type SimpleRDF = [Triple]

data Triple = Tri SimpleURI SimpleURI
      (Either SimpleURI Literal) deriving  (Eq,Ord)
data Literal = StrLit String | IntLit Integer | BoolLit Bool
      deriving (Eq, Ord)
```

The uri without prefix:

```
data SimpleURI = SimpleURI [Domain] [Domain] (Maybe Tag)
      deriving (Eq, Ord)
```

The process of excluding grammar sugar and prefix relay on a globe environment to record prefix and related rui. At the same time, it needs to recode the triples which are already evaluated. So, we introduced a helper state

```
StateT Env (Writer SimpleRDF)
```

to help to execute our functions.

3 Other features

3.1 Error Message

To start by combining a number of tokens syntactically into a syntax tree.

If, on failure to construct the syntax tree, the contents of the first token are displayed For example, if you write in pr1.stql you have written

```
x=1 1
```

Then, after lexical parsing, you have successfully obtained 4 tokens

Then, during the syntax analysis phase, it turns out that It is not possible to generate a syntax tree from these tokens, so this function is called. The information it displays about the rows and columns helps you to find the approximate location of the errors

3.2 Domain Cutting

Since Our group was initially worried about the internal processing of uri, the caused a lot of trouble in writing subj, pred
For example:

```
subj = ("www":"cw":"org":[],"problem7":[],"#subject":[])

pred = ("www":"cw":"org":[],"problem7":[],"#predicate":[])
```

All 10 questions take uri as a whole and do not involve the logic of determining whether a domain and subdomain contain a fragment, so there is absolutely no need to represent uri as ([String], [String], Object)

Our group went to a lot of trouble to cut the URL into three parts: domain, subdomain and tag, and then cut the three parts into many pieces. It ended up not working at all.

4 Execution Model

Our language's execution model is made up of three stages: identifying token, parsing the tokens, and showing the the results of parsing. Our language includes 35 tokens which is related to keywords and operators.

When keywords such as **"case","else","end"**are found in the input file, they will be parsed to the next stage.

Suppose now that I want to run pr1.stql, My program reads the contents of preldue.stql first then reads the contents of pr1.stql Then it combines the two, passes them to the parser, parses the syntax tree and gives it to the interpreter.