Weekly Assignment 2

Advanced Programming 2014 @ DIKU

Martin Jørgensen
University of Copenhagen
Department of Computer Science
tzk173@alumni.ku.dk

Casper B. Hansen
University of Copenhagen
Department of Computer Science
fvx507@alumni.ku.dk

September 30, 2014

Abstract

A parser should be implemented for a domain specific language, describing curves and operations on them.

Tasks

Introduction

For the resubmission we have changed to a new parser library, instead of going with Parsec we switched to ReadP which proved to be easier to work with for our simple purposes.

Parsing

Partwise parsers

To allow for nore readable code the parser is split into different smaller parsers/methods. Topmost in the file we have a number of smaller convenience parsers such as `chrToken`, `strToken`, `number` and so forth. These are meant to catch and parse small components that are likely to be used by severeal other parsers.

The main parser is the one that parses "programs", it is called form `parseString` and will in turn call the rest of the parsers (indirectly of course, since it only really calls the `defs` parser itself. The code for the method can be seen in Figure 1.

```
-- Parse a program
prog :: ReadP [Def]
prog = do
  d <- defs
  eof
  return d
```

Figure 1: The implementation of the `prog` method which parses programs/lists of definitions. (../CurvySyntax2.hs)

This pattern is repeating through the whole parser combinator "tree" where a general parser will call smaller parsers on the sub components, for instance `prog` will call the `defs` parsers which calls the `def` parser and so on. The code can be seen in Figure 2.

```
-- Parse list of Defs
defs :: ReadP [Def]
defs = many def

-- Parse single Def
def :: ReadP Def
def = do
  iden <- ident
  _ <- chrToken '='
  ct <- curve
  defop (Def iden ct [])
```

Figure 2: Parse a list of definitions. (../CurvySyntax2.hs)

Parsers suffixed with "op" are parsers which handle "operators" for a different parser, for instance `expr` have `exprop` which will handle the operators in expressions. This can be seen in Figure 3

```haskell
-- Parse expressions.
expr :: ReadP Expr
expr = (do
           t <- term
           exprop t
       ) +++
       (do
           skipSpaces
           _ <- string "width"
           munch1 isSpace
           c <- curve
           return $ Width c
       ) +++
       (do
           skipSpaces
           _ <- string "height"
           munch1 isSpace
           c <- curve
           return $ Height c
       )

-- Parse expression operators.
-- Notice that multiplication is handled in "term" further up.
exprop :: Expr -> ReadP Expr
exprop val = (do
                 _ <- chrToken '+'
                 t <- term
                 exprop(Add val t)
             ) <++ return val
```

Figure 3: Parse expressions. (../CurvySyntax2.hs)

There is more code, but this shold give a look into how our code is constructed.

`parseString`

Uses readP to parse a string with the parsers defined earlier in the program.

```haskell
-- Parses a string into a program.
parseString :: String -> Either Error Program
parseString s = case opt of
  [] -> Left "Parser error."
  (x:_) -> Right (fst x)
  where opt = readP_to_S prog s
```

Figure 4: The implementation of the `parseString` method. (../CurvySyntax2.hs)

`parseFile`

`parseFile` was implemented with the suggestion from the assignment text and can be seen in Figure 5.

```
-- Reads and parses a file to a program.
parseFile :: FilePath -> IO (Either Error Program)
parseFile filename = fmap parseString $ readFile filename
```

Figure 5: The implementation of the `parseFile` method. (../CurvySyntax2.hs)

Testing

As before we did a number of just random testing, and then constructed a number of larger tests for the report. The tests and code for running them can be seen in Figure 6.

Testing is done simply by loading the module and running the `runTests` method. This will run 6 different tests and print the result like so:

```
0=OK
1=OK
2=OK
3=OK
4=OK
5=OK
```

If one of the tests fail it will read “FAIL” instead of “OK” in the list. The tests are designed so they test a number of things such as the precedence of operators, assignments and expressions.

```
-- Let the testing begin!
runTests :: IO ()
runTests = do
  -- The test lines will get a bit long, sorry about that :/
  test 0 "c = (0,0)" (Right [Def "c" (Single (Point (Const 0.0) (Const 0.0)))[]])

  test 1 "c = a where {a = (0,0.5) ++ (1,1)}" (Right [Def "c" (Id "a") [Def "a" (Connect (Single (Point (Const 0.0) (Const 0.5))) (Single (Point (Const 1.0) (Const 1.0)))) []]])

  test 2 "c = (0,0) ++ (5, 42.5)" (Right [Def "c" (Connect (Single (Point (Const 0.0) (Const 0.0))) (Single (Point (Const 5.0) (Const 42.5)))) []])

  test 3 "c = (1,1) rot (2+3) -> (4,5)" (Right [Def "c" (Translate (Rot (Single (Point (Const 1.0) (Const 1.0))) (Add (Const 2.0) (Const 3.0))) (Point (Const 4.0) (Const 5.0))) []])

  test 4 "c = (1+1, 2) ++ (3*3+3,3)" (Right [Def "c" (Connect (Single (Point (Add (Const 1.0) (Const 1.0)) (Const 2.0))) (Single (Point (Add (Mult (Const 3.0) (Const 3.0)) (Const 3.0)) (Const 3.0)))) []])

  test 5 "a = (1,2) ++ (3,4) ^ (5,6)" (Right [Def "a" (Connect (Single (Point (Const 1) (Const 2))) (Over (Single (Point (Const 3) (Const 4))) (Single (Point (Const 5) (Const 6))))) []])
    where
      test i inp exp =
        if testParse inp exp
        then putStrLn(show(i) ++ "=OK")
        else putStrLn(show(i) ++ "=FAIL")
        where
          testParse inp exp = (parseString inp) == (exp)
```

Figure 6: Method for running all our tests. (../CurvySyntax2.hs)