Basic Parser

Homework 3

Christopher Salinas | Gavin McGuire

Using the following two BASIC programs, we wrote a parser in Haskell to parse and print the program.

foo.bas

```
10 LET A = 2
20 LET B = 3
30 LET C = 4
40 PRINT A * (B + C)
50 END
```

test.bas

```
20 INPUT H
25 LET X = INT(RND(1)*H+1)
27 PRINT X
30 FOR I = 1 TO H
35 PRINT I
40 IF I = X THEN 60
50 NEXT I
60 END
```

Our program is written in `BASIC.hs`, but it currently uses two modules that we wrote, `BasicParser.hs` and `BasicData.hs`. In addition to these two files that we wrote, we use the `Parselib.hs` that was provided to us in the #homework Slack channel.

BASIC.hs

```haskell
{-------------------------------------------------------------------------------
 -
 - BASIC.hs
 -
 - Christopher Salinas
 - Gavin McGuire
 -
 - CS 456
 - Fall 2020
 - University of New Mexico
 -
 -------------------------------------------------------------------------------}

import BasicParser
import System.IO
import System.Environment

main = do
  args <- getArgs
  handle <- openFile (head args) ReadMode
  program <- parse' handle
```

```
  hClose handle
  putStrLn $ show program
```

BasicData.hs

```
{-------------------------------------------------------------------------------
 -
 - BasicData.hs
 -
 - Christopher Salinas
 - Gavin McGuire
 -
 - CS 456
 - Fall 2020
 - University of New Mexico
 -
-------------------------------------------------------------------------------}

module BasicData where
  data Expr = Id Char
            | Num { val :: Int }
            | Int' Expr
            | Rnd Expr
            | CompareExpr String Expr Expr
            | MultExpr Char Expr Expr
            | AddExpr Expr Expr
            | NegateExpr Expr
            | PowerExpr Expr Expr

  instance Show Expr where
    show (Id c) =  [c]
    show (Num n) = show n
    show (Int' e) = "INT(" ++ show e ++ ")"
    show (Rnd e) = "RND(" ++ show e ++ ")"
    show (MultExpr c lhs rhs) = show lhs ++ " " ++ [c] ++ " " ++ show rhs
    show (AddExpr lhs rhs) = show lhs ++ " + " ++ show rhs
    show (CompareExpr op lhs rhs) = show lhs++ " " ++ op ++ " " ++ show rhs
    show (NegateExpr e) = "- " ++ show e
    show (PowerExpr base power) = show base ++ " ^ " ++ show power

  data Statement = END
                 | FOR Expr Expr Expr
                 | LET Expr Expr
                 | NEXT [Expr]
                 | PRINT Expr
                 | IFTHEN Expr Int
                 | INPUT String [Expr]

  instance Show Statement where
    show END = "END"
    show (FOR id init lim) = "FOR " ++ show id ++ " = " ++ show init ++ " TO " ++ show lim
    show (LET lhs rhs) = "LET " ++ show lhs ++ " = " ++ show rhs
    show (NEXT exprs) = "NEXT" ++ showExprList exprs
    show (PRINT e) = "PRINT " ++ show e
    show (IFTHEN e line) = "IF " ++ show e ++ " THEN " ++ show line
    show (INPUT "" exprs) = "INPUT" ++ showExprList exprs
    show (INPUT str exprs) = "INPUT " ++ str ++ ";" ++ showExprList exprs

  showExprList exprs = concat $ fmap addSpace exprs
  addSpace expr = " " ++ show expr
```

BasicParser.hs

```
{-------------------------------------------------------------------------------
 -
 - BasicParser.hs
 -
 - Christopher Salinas
 - Gavin McGuire
 -
 - CS 456
 - Fall 2020
 - University of New Mexico
 -
 -------------------------------------------------------------------------------}

module BasicParser where
  import BasicData
  import Parselib
  import System.IO

  -- STATEMENT PARSERS
  stmt :: Parser Statement
  stmt = end +++ for' +++ ifthen +++ input +++ let' +++ next' +++ print'

  end :: Parser Statement
  end = do
    symb "END"
    return END

  let' :: Parser Statement
  let' = do
    symb "LET"
    var <- id'
    symb "="
    val <- expr
    return $ LET var val

  print' :: Parser Statement
  print' = do
    symb "PRINT"
    e <- expr
    return $ PRINT e

  input :: Parser Statement
  input = do
    symb "INPUT"
    ids <- many id'
    return $ INPUT [] ids


  ifthen :: Parser Statement
  ifthen = do
    symb "IF"
    e <- expr
    symb "THEN"
    n <- nat
    return $ IFTHEN e n

  for' :: Parser Statement
  for' = do
    symb "FOR"
    var <- id'
    symb "="
    val <- expr
    symb "TO"
    e <- expr
    return $ FOR var val e
```

```haskell
next' :: Parser Statement
next' = do
  symb "NEXT"
  ids <- many id'
  return $ NEXT ids

-- EXPRESSION PARSERS
expr :: Parser Expr
expr = compareExpr

compareExpr :: Parser Expr
compareExpr = equate +++ addExpr

equate :: Parser Expr
equate = do
  m <- addExpr
  symb "="
  a <- compareExpr
  return $ CompareExpr "=" m a

addExpr :: Parser Expr
addExpr = add +++ multExpr

add :: Parser Expr
add = do
  m <- multExpr
  symb "+" +++ symb "-"
  a <- expr
  return $ AddExpr m a

multExpr :: Parser Expr
multExpr = multTimes +++ multDiv +++ negateExpr

multTimes :: Parser Expr
multTimes = do
  n <- negateExpr
  symb "*"
  m <- expr
  return $ MultExpr '*' n m

multDiv :: Parser Expr
multDiv = do
  n <- negateExpr
  symb "/"
  m <- expr
  return $ MultExpr '/' n m

negateExpr :: Parser Expr
negateExpr = negate' +++ powerExpr

negate' :: Parser Expr
negate' = do
  symb "-"
  p <- expr
  return $ NegateExpr p

powerExpr :: Parser Expr
powerExpr = power +++ value

power :: Parser Expr
power = do
  v <- value
  symb "^"
  p <- expr
  return $ PowerExpr v p

value :: Parser Expr
value = parens +++ function +++ variable +++ constant
```

```
parens :: Parser Expr
parens = do
  symb "("
  e <- expr
  symb ")"
  return e

variable :: Parser Expr
variable = id'

id' :: Parser Expr
id' = do
  c <- token letter
  return $ Id c

function :: Parser Expr
function = int' +++ rnd

int' :: Parser Expr
int' = do
  symb "INT"
  e <- parens
  return $ Int' e

rnd :: Parser Expr
rnd  = do
  symb "RND"
  e <- parens
  return $ Rnd e

constant :: Parser Expr
constant = number

number :: Parser Expr
number = do
  n <- nat
  return $ Num n

-- PARSING FUNCTIONS
line :: Parser (Int, Statement)
line = do
  n <- number -- line number
  s <- stmt
  return (val n, s)

p :: String -> (Int, Statement)
p s = case apply line s of
  [(a, "")] -> a
  _ -> error "Parser error."

parse' :: Handle -> IO [(Int, Statement)]
parse' handle = do
  eof <- hIsEOF handle
  if eof then return []
  else do
    s <- hGetLine handle
    let stmt = p s
    stmts <- parse' handle
    return $ [stmt] ++ stmts
```

Results

Running the programs using yield the following results:

```
$ ./BASIC bas/foo.bas
[(10,LET A = 2),(20,LET B = 3),(30,LET C = 4),(40,PRINT A * B + C),(50,END)]
$ ./BASIC bas/test.bas
[(20,INPUT H),(25,LET X = INT(RND(1) * H + 1)),(27,PRINT X),(30,FOR I = 1 TO H),(35,PRINT I),(40,IF I = X THEN 60),
(50,NEXT I),(60,END)]
```

Note: BASIC programs are in a `bas` directory