SRC

Justin Chin

May 1, 2018

Parser

```
module Parser
  where

import Debug.Trace

import Control.Applicative

import Control.Monad
import Control.Conditional
import Control.Monad.State
import Control.Monad.Except
import Control.Monad.Identity

import Data.Char
import Text.Printf
import System.IO

import Lexer

import Token
import TokenType
import AST

-- ADT describing the state of a parser
data ParserState = ParserState
  { tokens :: [Token]
  , current :: Token
  , next :: Token
  , logs :: [String]
  , errors :: [ParserError]
  } deriving (Show)

-- ADT describing Parser errors
-- Constructor: ParserError
-- Values: The token itself, and the associated error message
```

```
data ParserError = ParserError Token String
  deriving (Show)

-- Defines synonym for type signature on right hand side
-- essentially "Parser" is a direct replacement for "ExceptT ParserError (StateT ParserState Ide
-- this defines the monad stack
type Parser = ExceptT ParserError (StateT ParserState Identity)

type Lexeme = String

parseIO :: [Token] -> Parser a -> IO (Maybe a)
parseIO tokens parser = either (\es -> mapM_ printError es >> return Nothing) (return . Just) (p
  where
    printError (ParserError token msg) =
        putStrLn $ "[line " ++ ((show . token_line) token) ++ "] Error at "  ++ (if' (token_type

-- Given a list of tokens and a Parser we either return a list of errors, or the abstract syntax
parse :: [Token] -> Parser a -> Either [ParserError] a
parse tokens parser = runParser (initializeState tokens) parser

-- Takes a ParserState and a Parser and returns either a list of ParserErrors or an abstract syn
runParser :: ParserState -> Parser a -> Either [ParserError] a
runParser state p =
  let (results, finalState) = runIdentity $ runStateT (runExceptT p) state
  in
    if null $ errors finalState
    then either (\e -> Left [e]) (Right) results
    else Left $ (reverse . errors) finalState

{-
ParserState { tokens = tokens; errors = [] }
-}
initializeState :: [Token] -> ParserState
initializeState tokens = ParserState tokens (head tokens) (head $ tail tokens) [] []

-- determines if a token in the token stream matches a TokenType and an associated Lexeme (Strin
match :: TokenType -> Lexeme -> String -> Parser Bool
match ttype lexeme err = do
  state <- get
  let
    cur_type = token_type $ current state
  let
    cur_lexeme = token_lexeme $ current state
  case (cur_type == ttype && cur_lexeme == lexeme) of
      True -> return True
      _ -> (peek >>= \t -> pError t err)

matchType :: TokenType -> Parser Bool
```

```
matchType ttype = do
  state <- get
  let
    cur_type = token_type $ current state
  case cur_type == ttype of
      True -> return True
      _ -> return False

-- if current token matches a TokenType and a Lexeme, advance the ParserState and return the tok
-- consume :: TokenType -> Lexeme -> String -> Parser Token
consume ttype lexeme err = do
  cur <- peek
  -- trace ("Token: " ++ (show $ token_type cur) ++ " \t " ++ "Lexeme: " ++ (show $ token_lexeme
  ifM (match ttype lexeme err) (advance >> return peek) (peek >>= \t -> pError t err)

consumeType ttype err = do
  cur <- peek
  -- trace ("Token: " ++ (show $ token_type cur) ++ " \t " ++ "Lexeme: " ++ (show $ token_lexeme
  ifM (matchType ttype) (advance >> return cur) (peek >>= \t -> pError t err)

-- returns the current token in the stream
peek :: Parser Token
peek = do
  s <- get
  return (current s)

lookahead :: Parser Token
lookahead = do
  s <- get
  return (next s)

pError :: Token -> String -> Parser a
pError token message = throwError $ ParserError token message

-- advances the ParserState
-- advance ‘gets‘ the current ParserState, and uses it as an argument to the anonymous function
-- which takes the current state, and overwrites it with the resulting state of advancing

advance = get >>= \state -> do
  cur <- peek
  traceM("Token: " ++ (show $ token_type cur) ++ " \t " ++ "Lexeme: " ++ (show $ token_lexeme cu
  put state { tokens = tail $ tokens state
            , current = next state
            , logs = ("Token: " ++ (show $ token_type cur) ++ " \t " ++ "Lexeme: " ++ (show $ to
            , next = head $ tail $ tail (tokens state) }

-- gets the current ParserState and overwrites prepends the new error to the error list
handleParseError :: ParserError -> Parser ()
```

```
handleParseError err = do
  state <- get
  put state { errors = err : errors state }

-- Entry point into the recursive descent
parseRat18S :: Parser Rat18S
parseRat18S = do
  traceM("\t<Rat18S> ::= <OptFunctionDefinitions> %% <OptDeclarationlist> <StatementList>")
  defs <- parseOptFunctionDefs
  consume EndOfDefs "%%" "Expecting '%%' after function definitions."
  decs <- parseOptDeclarationList
  stmts <- parseStatementList
  return (Rat18S defs decs stmts)

parseOptFunctionDefs :: Parser OptFunctionDefinitions
parseOptFunctionDefs = do
  traceM("\t<OptFunctionDefinitions ::= <FunctionDefinitions> | <Empty>")
  cur <- peek
  case cur of
    Token Keyword "function" _ -> do
      defs <- parseFunctionDefs
      return (OptFunctionDefinitions defs)
    _ -> return (EmptyDefs Empty)

parseFunctionDefs :: Parser FunctionDefinitions
parseFunctionDefs = do
  traceM("\t<FunctionDefitions> ::= <Function> <FDPrime>")
  def <- parseFunction
  defsprime <- parseFDPrime
  return (FunctionDefinitions def defsprime)

parseFunction :: Parser Function
parseFunction = do
  consume Keyword "function" "Expecting keyword 'function' in function definition."
  traceM("\t<Function> ::= function <Identifier> [ <OptParameterList> ] <OptDeclarationList> <Bo
  id <- parseIdentifier
  -- traceM("\t<Identifier> ::= id | <Integer> | <Real>")
  cur <- peek
  case cur of
    Token LBracket _ _ -> do
      consumeType LBracket "Expecting '[' before optional paramater list."
      params <- parseOptParameterList
      consumeType RBracket "Expecting ']' after optional parameter list."
      decs <- parseOptDeclarationList
      body <- parseBody
      return (Function id params decs body)
    _ -> do
      decs <- parseOptDeclarationList
```

```
      body <- parseBody
      return (Function id (EmptyParamList Empty) decs body)

parseFDPrime :: Parser FDPrime
parseFDPrime = do
  cur <- peek
  case cur of
    Token Keyword "function" _ -> do
      traceM("\t<FDPrime> ::= <FunctionDefinitions>")
      defs <- parseFunctionDefs
      return (FDPrime defs)
    _ -> do
      traceM("\t<FDPrime> ::= <Empty>")
      return (EmptyFDPrime Empty)

parseOptParameterList :: Parser OptParameterList
parseOptParameterList = do
  traceM("\t<OptParameterList> ::= <ParamaterList> | <Empty>")
  params <- parseParameterList
  return (OptParameterList params)

parseParameterList :: Parser ParameterList
parseParameterList = do
  traceM("\t<ParamaterList> ::= <Parameter> <ParameterListPrime>")
  param <- parseParameter
  paramprime <- parsePLPrime
  return (ParameterList param paramprime)

parseParameter :: Parser Parameter
parseParameter = do
  traceM("\t<Parameter> ::= <IDs> : <Qualifier>")
  id <- parseID
  consumeType Colon "Expecting ':' between identifier and qualifier in parameter list."
  quals <- parseQualifier
  return (Parameter1 id quals)

parsePLPrime :: Parser PLPrime
parsePLPrime = do
  cur <- peek
  case token_type cur of
    Comma -> do
      traceM("\t<ParameterListPrime> ::= <ParameterList>")
      consumeType Comma "Expecting ',' between paramaters"
      param <- parseParameterList
      return (PLPrime param)
    _ -> do
      traceM("\t<ParameterListPrime> ::= <Empty>")
      return (PLPrimeEmpty Empty)
```

```
parseOptDeclarationList :: Parser OptDeclarationList
parseOptDeclarationList = do
  cur <- peek
  case cur of
    Token Keyword "int" _ -> do
      traceM("\t<OptDeclarationList> ::= <DeclarationList>")
      decs <- parseDeclarationList
      return (OptDeclarationList decs)
    Token Keyword "real" _ -> do
      traceM("\t<OptDeclarationList> ::= <DeclarationList>")
      decs <- parseDeclarationList
      return (OptDeclarationList decs)
    Token Keyword "boolean" _ -> do
      traceM("\t<OptDeclarationList> ::= <DeclarationList>")
      decs <- parseDeclarationList
      return (OptDeclarationList decs)
    _ -> do
      traceM("\t<OptDeclarationList> ::= <Empty>")
      return (EmptyDecs Empty)

parseDeclarationList :: Parser DeclarationList
parseDeclarationList = do
  traceM("\t<DeclarationList> ::= <Declaration> <DeclarationListPrime>")
  dec <- parseDeclaration
  decprime <- parseDLPrime
  return (DeclarationList dec decprime)

parseDLPrime :: Parser DLPrime
parseDLPrime = do
  cur <- peek
  case cur of
    Token Semicolon _ _ -> do
      traceM("\t<DeclarationListPrime> ::= <Empty>")
      return (DLPrimeEmpty Empty)
    Token Keyword "int" _ -> do
      traceM("\t<DeclarationListPrime> ::= <DeclarationList>")
      decs <- parseDeclarationList
      return (DLPrime decs)
    Token Keyword "boolean" _ -> do
      traceM("\t<DeclarationListPrime ::= <DeclarationList>")
      decs <- parseDeclarationList
      return (DLPrime decs)
    Token Keyword "real" _ -> do
      traceM("\t<DeclarationListPrime> ::= <DeclarationList>")
      decs <- parseDeclarationList
      return (DLPrime decs)
    _ -> do
```

```
      traceM("\t<DeclarationListPrime> ::= <Empty>")
      return (DLPrimeEmpty Empty)

parseDeclaration :: Parser Declaration
parseDeclaration = do
  traceM("\t<Declaration> ::= <Qualifier> <IDs>")
  qual <- parseQualifier
  id <- parseID
  return (Declaration1 qual id)

parseQualifier :: Parser Qualifier
parseQualifier = do
  cur <- peek
  case cur of
    Token Keyword "int" _ -> do
      advance
      traceM("\t<Qualifier> ::= int")
      return QualifierInt
    Token Keyword "boolean" _ -> do
      advance
      traceM("\t<Qualifier> ::= boolean")
      return QualifierBoolean
    Token Keyword "real" _ -> do
      advance
      traceM("\t<Qualifier> ::= real")
      return QualifierReal
    _ -> peek >>= \e -> pError e "Expecting one of int, boolean, real."

parseStatementList :: Parser StatementList
parseStatementList = do
  traceM("\t<StatementList> ::= <Statement> <StatementListPrime>")
  stmt <- parseStatement
  stmtprime <- parseSLPrime
  return (StatementList stmt stmtprime)

parseStatement :: Parser Statement
parseStatement = do
  cur <- peek
  case cur of
    Token LBrace _ _ -> do
      traceM("\t<Statement> ::= <Compound>")
      compound <- parseCompound
      return (StatementCompound compound)
    Token Keyword "if" _ -> do
      traceM("\t<Statement> ::= <If>")
      ifexpr <- parseIf
      return (StatementIf ifexpr)
    Token Keyword "while" _ -> do
```

```
      traceM("\t<Statement> ::= <While>")
      while <- parseWhile
      return (StatementWhile while)
    Token Keyword "get" _ -> do
      traceM("\t<Statement> ::= <Scan>")
      scan <- parseScan
      return (StatementScan scan)
    Token Keyword "put" _ -> do
      traceM("\t<Statement> ::= <Print>")
      printexpr <- parsePrint
      return (StatementPrint printexpr)
    Token Keyword "return" _ -> do
      traceM("\t<Statement> ::= <Return>")
      ret <- parseReturn
      return (StatementReturn ret)
    Token Identifier _ _ -> do
      traceM("\t<Statement> ::= <Assign>")
      assign <- parseAssign
      return (StatementAssign assign)
    _ -> peek >>= \t -> pError t "Unexpected token in statement"

parseSLPrime :: Parser SLPrime
parseSLPrime = do
  cur <- peek
  case cur of
    Token LBrace _ _ -> do
      traceM("\t<StatementListPrime> ::= <StatementList>")
      stmtlst <- parseStatementList
      return (SLPrime stmtlst)
    Token Keyword "if" _ -> do
      traceM("\t<StatementListPrime> ::= <StatementList>")
      stmtlst <- parseStatementList
      return (SLPrime stmtlst)
    Token Keyword "while" _ -> do
      traceM("\t<StatementListPrime> ::= <StatementList>")
      stmtlst <- parseStatementList
      return (SLPrime stmtlst)
    Token Keyword "scan" _ -> do
      traceM("\t<StatementListPrime> ::= <StatementList>")
      stmtlst <- parseStatementList
      return (SLPrime stmtlst)
    Token Keyword "put" _ -> do
      traceM("\t<StatementListPrime> ::= <StatementList>")
      stmtlst <- parseStatementList
      return (SLPrime stmtlst)
    Token Keyword "get" _ -> do
      traceM("\t<StatementListPrime> ::= <StatementList>")
      stmtlst <- parseStatementList
```

```
      return (SLPrime stmtlst)
    Token Keyword "return" _ -> do
      traceM("\t<StatementListPrime> ::= <StatementList>")
      stmtlst <- parseStatementList
      return (SLPrime stmtlst)
    Token Identifier _ _ -> do
      traceM("\t<StatementListPrime> ::= <StatementList>")
      stmtlst <- parseStatementList
      return (SLPrime stmtlst)
    _ -> do
      traceM("\t<StatementListPrime> ::= <Empty>")
      return (SLPrimeEmpty Empty)

parseID :: Parser IDs
parseID = do
  id <- parseIdentifier
  traceM("\t<IDs> :: = <Identifier> <IDsPrime>")
  idprime <- parseIDPrime
  return (IDs id idprime)

parseIDPrime :: Parser IDsPrime
parseIDPrime = do
  -- traceM("\t<IDsPrime> ::= , <IDs> | <Empty>")
  cur <- peek
  case token_type cur of
    Comma -> do
      consume Comma "," "Expecting ’,’."
      traceM("\t<IDsPrime> ::= , <IDs>")
      id <- parseID
      return (IDsPrime id)
    Colon -> do
      traceM("\t<IDsPrime> ::= <Empty>")
      return (IDsPrimeEmpty Empty)
    Semicolon -> do
      traceM("\t<IDsPrime> ::= <Empty>")
      consume Semicolon ";" "Expecting ’;’"
      return (IDsPrimeEmpty Empty)
    RParen -> do
      traceM("\t<IDsPrime> ::= <Empty>")
      return (IDsPrimeEmpty Empty)
    _ -> (peek >>= \t -> pError t "Unexpected token in IDs.")

parseBody :: Parser Body
parseBody = do
  _ <- consume LBrace "{" "Expecting ’{’ before statement list."
  traceM("\t<Body> ::= { <StatementList> }")
  stmts <- parseStatementList
  _ <- consume RBrace "}" "Expecting ’}’ after statement list."
```

```
  return (Body stmts)

parseCondition :: Parser Condition
parseCondition = do
  traceM("\t<Condition> ::= <Expression> <Relop> <Expression>")
  expr <- parseExpression
  relop <- parseRelop
  expr2 <- parseExpression
  return (Condition expr relop expr2)

parseRelop :: Parser Relop
parseRelop = do
  cur <- peek
  case cur of
    Token Greater _ _ -> do
      advance
      traceM("\t<Relop> ::= >")
      return (Relop cur)
    Token Less _ _ -> do
      advance
      traceM("\t<Relop> ::= <")
      return (Relop cur)
    Token EGT _ _ -> do
      advance
      traceM("\t<Relop> ::= =>")
      return (Relop cur)
    Token ELT _ _ -> do
      advance
      traceM("\t<Relop> ::= =<")
      return (Relop cur)
    Token Equals _ _ -> do
      advance
      traceM("\t<Relop> ::= ==")
      return (Relop cur)
    Token NEquals _ _ -> do
      advance
      traceM("\t<Relop> ::= ^=")
      return (Relop cur)

parseExpression :: Parser Expression
parseExpression = do
  traceM("\t<Expression> ::= <Term> <ExpressionPrime>")
  term <- parseTerm
  expprime <- parseExpressionPrime
  return (Expression term expprime)

parseExpressionPrime :: Parser EPrime
parseExpressionPrime = do
```

```
  cur <- peek
  case cur of
    Token Plus _ _ -> do
      consumeType Plus "Expecting '+' in expression."
      traceM("\t<ExpressionPrime> ::= + <Term> <ExpressionPrime>")
      term <- parseTerm
      expprime <- parseExpressionPrime
      return (EPrimePlus term expprime)
    Token Minus _ _-> do
      consumeType Minus "Expecting '-' in expression."
      traceM("\t<ExpressionPrime> ::= - <Term> <ExpressionPrime>")
      term <- parseTerm
      expprime <- parseExpressionPrime
      return (EPrimeMinus term expprime)
    _ -> do
      traceM("\t<ExpressionPrime> ::= <Empty>")
      return (EPrime Empty)

parseTerm :: Parser Term
parseTerm = do
  traceM("\t<Term> ::= <Factor> <TermPrime>")
  fact <- parseFactor
  tprime <- parseTermPrime
  return (Term fact tprime)

parseTermPrime :: Parser TermPrime
parseTermPrime = do
  cur <- peek
  case cur of
    Token Times _ _ -> do
      consumeType Times "Expecting '*'."
      traceM("\t<TermPrime> ::= * <Factor> <TermPrime>")
      factor <- parseFactor
      tprime <- parseTermPrime
      return (TermPrimeMult factor tprime)
    Token Div _ _ -> do
      consumeType Div "Expecting '/'."
      traceM("\t<TermPrime> ::= / <Factor> <TermPrime>")
      factor <- parseFactor
      tprime <- parseTermPrime
      return (TermPrimeDiv factor tprime)
    _ -> do
      traceM("\t<TermPrime> ::= <Empty>")
      return (TermPrime Empty)

parseFactor :: Parser Factor
parseFactor = do
  traceM("\t<Factor> ::= - <Primary> | <Primary>")
```

```
  prim <- parsePrimary
  return (FactorPrimary prim)

parsePrimary :: Parser Primary
parsePrimary = do
  cur <- peek
  next <- lookahead
  case token_type cur of
    LParen -> do
      traceM("\t<Primary> ::= ( <Expression> )")
      consumeType LParen "Expecting '(' before expression."
      expr <- parseExpression
      consumeType RParen "Expecting ')' after expression."
      return (Expr expr)
    Identifier -> do
      case token_type next of
        LParen -> do
          traceM("\t<Primary> ::= <Identifier> ( <IDs> )")
          ident <- parseIdentifier
          consumeType LParen "Expecting '(' before function arguments."
          args <- parseID
          consumeType RParen "Expecting ')' after function arguments."
          return (Call (Ident ident) args)
        _ -> do
          advance
          traceM("\t<Primary> ::= <Identifier>")
          return (Id (Ident cur))
    _ -> do
      cur <- peek
      advance
      case cur of
        Token Int n _ -> do
          traceM("\t<Primary> ::= <Integer>")
          return (Integer (read n))
        Token Real r _ -> do
          traceM("\t<Primary> ::= <Real>")
          return (Double (read r))
        Token Keyword "true" _ -> do
          traceM("\t<Primary> ::= true")
          return (BoolTrue)
        Token Keyword "false" _ -> do
          traceM("\t<Primary> ::= false")
          return (BoolFalse)

        -- Token EOF _ _ -> peek >>= \t -> pError t "Unexpected end of file"

parseIdentifier :: Parser Token
parseIdentifier = do
```

```
  consumeType Identifier "parseIdentifier: Expecting identifier."

parseIf :: Parser If
parseIf = do
  consume Keyword "if" "Expecting keyword 'if' in If statement."
  traceM("\t<If> ::= if ( <Condition> ) <Statement> endif | if ( <Condition> ) else <Statement>
  consume LParen "(" "Expecting '(' in if-expression."
  cond <- parseCondition
  _ <- consume RParen ")" "Expecting ')' in if-expression."
  stmt <- parseStatement
  next <- peek
  case next of
    Token Keyword "else" _ -> do
      consume Keyword "else" "Expecting keyword 'else' in If-Else statement"
      stmt2 <- parseStatement
      consume Keyword "endif" "Expecting keyword 'endif'."
      return (IfElseIf cond stmt stmt2)
    _ -> do
      consume Keyword "endif" "Expecting keyword 'endif'."
      return (IfElse cond stmt)

parseReturn :: Parser Return
parseReturn = do
  consume Keyword "return" "Expecting keyword 'return'."
  traceM("\t<Return> ::= return <ReturnPrime>")
  next <- peek
  case next of
    Token Semicolon _ _ -> do
      consumeType Semicolon "Expecting ';' at end of return statement"
      traceM("\t<ReturnPrime> ::= ;")
      return (Return (RPrime Empty))
    _ -> do
      traceM("\t<ReturnPrime> ::= <Expression> ;")
      expr <- parseExpression
      consumeType Semicolon "Expecting ';' at end of return statement"
      return (Return $ RPrimeExp expr)

parsePrint :: Parser Print
parsePrint = do
  consume Keyword "put" "Expecting keyword 'put'."
  traceM("\t<Print> ::= put ( <Expression> ) ;")
  consume LParen "(" "Expecting '(' before expression."
  expr <- parseExpression
  consume RParen ")" "Expecting ')' at end of expression."
  consumeType Semicolon "Expecting ';' at end of print statement."
  return (Print expr)

parseScan :: Parser Scan
```

```
parseScan = do
  _ <- consume Keyword "get" "Expecting keyword 'get'."
  traceM("\t<Scan> ::= get ( <IDs> ) ;")
  _ <- consume LParen "(" "Expecting '('"
  ids <- parseID
  _ <- consume RParen ")" "Expecting ')'."
  _ <- consume Semicolon ";" "Expecting ';' at end of statement."
  return (Scan ids)

parseWhile :: Parser While
parseWhile = do
  _ <- consume Keyword "while" "Expecting keyword while."
  traceM("\t<While> ::= while ( <Condition> ) <Statement>")
  _ <- consume LParen "(" "Expecting '('."
  cond <- parseCondition
  _ <- consume RParen ")" "Expecting ')'."
  stmt <- parseStatement
  return (While cond stmt)

parseCompound :: Parser Compound
parseCompound = do
  consume LBrace "{" "Expecting '{'."
  traceM("\t<Compound> ::= { <StatementList> }")
  stmts <- parseStatementList
  consume RBrace "}" "Expecting '}'."
  return $ Compound stmts

parseAssign :: Parser Assign
parseAssign = do
  ident <- parseIdentifier
  traceM("\t<Assign> ::= <Identifier> = <Expression> ;")
  consume TokenType.Assign "=" "Expecting '='."
  expr <- parseExpression
  consume Semicolon ";" "Expecting ';'."
  return $ AST.Assign ident expr

parseEmpty :: Parser Empty
parseEmpty = return Empty

-- prettyParse p :: Parser a -> IO()
-- prettyParse p =
```

AST

```
module AST where

import Token
```

```
import Lexer

newtype Ident = Ident Token deriving (Eq, Ord, Show)

data Empty = Empty
  deriving (Eq, Ord, Show)

data Rat18S
    = Rat18S OptFunctionDefinitions OptDeclarationList StatementList
  deriving (Eq, Ord, Show)

data OptFunctionDefinitions
    = OptFunctionDefinitions FunctionDefinitions
    | EmptyDefs Empty
  deriving (Eq, Ord, Show)

data FunctionDefinitions = FunctionDefinitions Function FDPrime
  deriving (Eq, Ord, Show)

data FDPrime
    = FDPrime FunctionDefinitions
    | EmptyFDPrime Empty
  deriving (Eq, Ord, Show)

data Function
    = Function Token OptParameterList OptDeclarationList Body
  deriving (Eq, Ord, Show)

data OptParameterList
    = OptParameterList ParameterList
    | EmptyParamList Empty
  deriving (Eq, Ord, Show)

data ParameterList = ParameterList Parameter PLPrime
  deriving (Eq, Ord, Show)

data PLPrime
    = PLPrime ParameterList
    | PLPrimeEmpty Empty
  deriving (Eq, Ord, Show)

data Parameter = Parameter1 IDs Qualifier
  deriving (Eq, Ord, Show)

data Qualifier = QualifierInt | QualifierBoolean | QualifierReal
  deriving (Eq, Ord, Show)

data Body = Body StatementList
```

```
  deriving (Eq, Ord, Show)

data OptDeclarationList
    = OptDeclarationList DeclarationList
    | EmptyDecs Empty
  deriving (Eq, Ord, Show)

data DeclarationList = DeclarationList Declaration DLPrime
  deriving (Eq, Ord, Show)

data DLPrime
    = DLPrime DeclarationList | DLPrimeEmpty Empty
  deriving (Eq, Ord, Show)

data Declaration = Declaration1 Qualifier IDs
  deriving (Eq, Ord, Show)

data IDs = IDs Token IDsPrime
  deriving (Eq, Ord, Show)

data IDsPrime = IDsPrime IDs | IDsPrimeEmpty Empty
  deriving (Eq, Ord, Show)

data StatementList = StatementList Statement SLPrime
  deriving (Eq, Ord, Show)

data SLPrime
    = SLPrime StatementList | SLPrimeEmpty Empty
  deriving (Eq, Ord, Show)

data Statement
    = StatementCompound Compound
    | StatementAssign Assign
    | StatementIf If
    | StatementReturn Return
    | StatementPrint Print
    | StatementScan Scan
    | StatementWhile While
  deriving (Eq, Ord, Show)

data Compound = Compound StatementList
  deriving (Eq, Ord, Show)

data Assign = Assign Token Expression
  deriving (Eq, Ord, Show)

data If
    = IfElse Condition Statement | IfElseIf Condition Statement Statement
```

```
  deriving (Eq, Ord, Show)

data Return = Return RPrime
  deriving (Eq, Ord, Show)

data RPrime = RPrime Empty | RPrimeExp Expression
  deriving (Eq, Ord, Show)

data Print = Print Expression
  deriving (Eq, Ord, Show)

data Scan = Scan IDs
  deriving (Eq, Ord, Show)

data While = While Condition Statement
  deriving (Eq, Ord, Show)

data Condition = Condition Expression Relop Expression
  deriving (Eq, Ord, Show)

data Relop = Relop Token
  deriving (Eq, Ord, Show)

data Expression = Expression Term EPrime
  deriving (Eq, Ord, Show)

data EPrime
    = EPrimePlus Term EPrime | EPrimeMinus Term EPrime | EPrime Empty
  deriving (Eq, Ord, Show)

data Term = Term Factor TermPrime
  deriving (Eq, Ord, Show)

data TermPrime
    = TermPrimeMult Factor TermPrime
    | TermPrimeDiv Factor TermPrime
    | TermPrime Empty
  deriving (Eq, Ord, Show)

data Factor = Factor1 Primary | FactorPrimary Primary
  deriving (Eq, Ord, Show)

data Primary
    = Id Ident
    | Integer Integer
    | Call Ident IDs
    | Expr Expression
    | Double Double
```

```
    | BoolTrue
    | BoolFalse
  deriving (Eq, Ord, Show)
```

Lexer

```
module Lexer
  where

import Data.Char
import Text.Printf
import Token
import TokenType

-- function mapping a character to an operator
operator ::  Char -> TokenType
operator tt | tt == '+'  = Plus
            | tt == '-'  = Minus
            | tt == '*'  = Times
            | tt == '/'  = Div
            | tt == '>'  = Greater
            | tt == '<'  = Less

-- function mapping a character to a separator
separator :: Char -> TokenType
separator sep | sep == '(' = LParen
              | sep == ')' = RParen
              | sep == '{' = LBrace
              | sep == '}' = RBrace
              | sep == '[' = LBracket
              | sep == ']' = RBracket
              | sep == ':' = Colon
              | sep == ';' = Semicolon
              | sep == ',' = Comma

-- define some lists
operators  = "+-*/><"

separators = "(){}[]:;,"

keywords   = ["function","return",
              "int","boolean","real",
              "if","else","endif",
              "put","get","while",
              "true","false"]

-- Match identifiers against keyword list
```

```
kwLookup :: Int -> String -> Token
kwLookup line str
  | str `elem` keywords = Token { token_type = Keyword
                                , token_lexeme = str
                                , token_line = line }
  | otherwise = Token{ token_type = Identifier
                     , token_lexeme = str
                     , token_line = line }

lexer :: String -> [Token]
lexer input = lexer1 1 (input ++ " ")              {- concat whitespace at end of input
                                                    to prevent EOF from ending a token -}

-- hack to kind of add line numbers to tokens by passing it as an argument through the execution
-- should go back at some point and figure out how to encapsulate this process in a state monad
-- more idiomatic approach, but this will have to do more now since we need line numbers for err
-- reporting in the parser

lexer1 :: Int -> String -> [Token]                    -- recursive driving function for the le
lexer1 line [] = [Token EOF "EOF" line]                                  -- base case
lexer1 line input =
  let
    (token,remaining) = dfsa line 0 "" input          -- start machine in state 0
  in
    case token_type token of
      Whitespace -> lexer1 line remaining
      Newline -> lexer1 (line + 1) remaining
      _ -> token : lexer1 line remaining

{-
    From some state, build a string of characters from input
    until a token is found, returning a pair
-}
dfsa :: Int -> Integer -> String -> String -> (Token,String)
dfsa line state currTokStr []     = (Token { token_type = UnexpectedEOF
                                           , token_lexeme = currTokStr
                                           , token_line = line }, "")
dfsa line state currTokStr (c:cs) =
  let
    (nextState,   isConsumed)      = getNextState state c
    (nextTokStr,  remaining)       = nextStrings currTokStr c cs isConsumed
    (isAccepting, token)           = accepting nextState line nextTokStr
  in
    if isAccepting
    then (token, remaining)
    else dfsa line nextState nextTokStr remaining

nextStrings :: String -> Char -> String -> Bool -> (String,String)
```

```
nextStrings tokStr c remaining isConsumed
  | isConsumed     = (tokStr ++ [c], remaining)
  | not isConsumed = (tokStr       , c:remaining)           -- cons unconsumed char onto remaini

charToString :: Char -> String
charToString c = [c]

-- Define accepting states for the machine
accepting :: Integer -> Int -> String -> (Bool,Token)

accepting 2 line currTokStr  = (True, (kwLookup line currTokStr))       -- Identifiers/Keywords
accepting 3 line currTokStr  = (True, Token { token_type = Identifier
                                            , token_lexeme = currTokStr
                                            , token_line = line })

accepting 12 line currTokStr = (True, Token { token_type = Int
                                            , token_lexeme = show $ (read currTokStr :: Int)
                                            , token_line = line }) -- Integers/Reals
accepting 13 line currTokStr = (True, Token { token_type = Real
                                            , token_lexeme = show $ (read currTokStr :: Double)
                                            , token_line = line })

accepting 20 line (x:xs) = (True, Token { token_type = operator x
                                        , token_lexeme = charToString x
                                        , token_line = line })
accepting 22 line _      = (True, Token { token_type = NEquals
                                        , token_lexeme = "^="
                                        , token_line = line })
accepting 24 line _      = (True, Token { token_type = Equals
                                        , token_lexeme = "=="
                                        , token_line = line })
accepting 25 line _      = (True, Token { token_type = ELT
                                        , token_lexeme = "<="
                                        , token_line = line })
accepting 26 line _      = (True, Token { token_type = EGT
                                        , token_lexeme = "=>"
                                        , token_line = line })
accepting 27 line _      = (True, Token { token_type = Assign
                                        , token_lexeme = "="
                                        , token_line = line })

accepting 30 line (x:xs) = (True, Token { token_type = separator x
                                        , token_lexeme = charToString x
                                        , token_line = line }) -- Separator
accepting 32 line _      = (True, Token { token_type = EndOfDefs
                                        , token_lexeme = "%%"
                                        , token_line = line })
```

```
accepting 51 line _ = (True, Token { token_type = Whitespace
                                   , token_lexeme = ""
                                   , token_line = line })                          -- Comment, tre

accepting 97 line _ = (True, Token { token_type = Newline
                                   , token_lexeme = ""
                                   , token_line = line })                             -- Newline,
accepting 98 line _ = (True, Token { token_type = Whitespace
                                   , token_lexeme = ""
                                   , token_line = line })                          -- Whitespace

accepting 100 line currTokStr = (True, Token { token_type = Unknown
                                             , token_lexeme = currTokStr
                                             , token_line = line })
accepting _ line currTokStr   = (False, Token { token_type = Unknown
                                              , token_lexeme = currTokStr
                                              , token_line = line }) -- all other states are non

getNextState :: Integer -> Char -> (Integer,Bool) -- Deterministically run machine to next state
getNextState 0 c
  | c `elem` separators = (30, True)      -- separator
  | c `elem` operators  = (20, True)      -- singleton operators
  | c == '%'            = (31, True)      -- end of function definitions
  | c == '^'            = (21, True)      -- beginning not equals
  | c == '='            = (23, True)      -- beginning of rest of relop
  | c == '!'            = (50, True)      -- beginning of comment
  | isLetter c          =  (1, True)      -- in id/keyword
  | isDigit  c          = (10, True)      -- in number
  | c == '\n'           = (97, True)      -- newline, increment line counter
  | isSpace  c          = (98, True)      -- whitespace final state
  | otherwise           = (99, True)      -- error

-- Idents/Keywords
getNextState 1 c
  | c == '$'      = (3, True)    -- ends an identifier
  | isLetter c    = (1, True)    -- accept any number of letters
  | isDigit  c    = (4, True)    -- accept any number of digits
  | otherwise     = (2, False)   -- non-identifier character, do not consume

-- Digit in ident/keyword
getNextState 4 c
  | c == '$'      = (3, True)
  | isDigit c     = (4, True)
  | isLetter c    = (1, True)
  | otherwise     = (99, False)

-- Numbers
getNextState 10 c
```

```
  | isDigit c     = (10, True)   -- accept any number of digits
  | c == '.'      = (11, True)   -- floating point number
  | otherwise     = (12, False)  -- non-digit, do not consume

getNextState 11 c
  | isDigit c     = (11, True)   -- continue floating point number
  | otherwise     = (13, False)  -- non-digit, do not consume

-- Operators
getNextState 21 c
  | c == '='      = (22, True)   -- NEquals
  | otherwise     = (99, False)  -- Unknown character

getNextState 23 c
  | c == '='      = (24, True)   -- Equals
  | c == '<'      = (25, True)   -- ELT
  | c == '>'      = (26, True)   -- EGT
  | otherwise     = (27, False)  -- Assign

getNextState 31 c
  | c == '%'      = (32, True)
  | otherwise     = (100, False)

-- Comments
getNextState 50 c
  | c == '!'      = (51, True)   -- End of comment
  | otherwise     = (50, True)

getNextState 99 c
  | isSpace c     = (100, False)
  | otherwise     = (99, True)

getNextState _ _   = (99, True)  -- Error, catch-all patterns not matching those defined above

-- helper functions to print Tokens relying on pattern matching
showTokenType :: Token -> String
showTokenType token = show $ token_type token

showTokenLexeme :: Token -> String
showTokenLexeme token = token_lexeme token

showTokenLineNumber :: Token -> String
showTokenLineNumber token = (show $ token_line token)


prettyPrint :: [Token] -> IO ()
prettyPrint [] = printf ""
prettyPrint (t:ts) =
```

```
  let
    token = showTokenType t
    lexeme = showTokenLexeme t
    line = showTokenLineNumber t
  in
    do
      printf "%12s %12s %12s\n" token lexeme line
      prettyPrint ts

prettyPrint1 :: Token -> IO ()
prettyPrint1 t =
  let
    token = showTokenType t
    lexeme = showTokenLexeme t
    line = showTokenLineNumber t
  in
    do
      printf "%12s %12s %12s\n" token lexeme line
```

TokenType

```
module TokenType where

data TokenType = Identifier
               | Keyword
               | Int
               | Real
               | RParen
               | LParen
               | LBrace
               | RBrace
               | LBracket
               | RBracket
               | Colon
               | Semicolon
               | Comma
               | EndOfDefs
               | Plus
               | Minus
               | Times
               | Div
               | Greater
               | Less
               | EGT
               | ELT
               | Assign
               | Equals
```

```
              | NEquals
              | Whitespace
              | Newline
              | UnexpectedEOF
              | Unknown
              | EOF
              deriving (Show, Eq, Ord, Read)
```

Token

```
module Token where

import TokenType

data Token = Token
  { token_type :: TokenType
  , token_lexeme :: String
  , token_line :: Int
  } deriving (Show, Eq, Ord)
```