```
-- setting the "warn-incomplete-patterns" flag asks GHC to warn you
-- about possible missing cases in pattern-matching definitions
{-# OPTIONS_GHC -fwarn-incomplete-patterns #-}

-- see https://wiki.haskell.org/Safe_Haskell
{-# LANGUAGE Safe #-}

module Parser (parseProgram) where

import Data.Char
import Control.Monad

import AbstractSyntax
import Parsing

program :: Parser Program

expr, expr1, expr2, expr3, expr4, expr5, expr6, expr7 :: Parser Expr

orOp, andOp, eqOp, compOp, addOp, mulOp, notOp :: Parser ([Expr] -> Expr)

undefinedParse :: Parser a
undefinedParse = P (\ _ -> [])

program =
      assignment
  <|> block
  <|> whileStatement
  <|> ifStatement
  <|> readStatement
  <|> writeStatement
  <|> printStatement
  <|> forStatement

assignment =
  do
    i <- identif
    symbol ":="
    e <- expr
    symbol ";"
    return (i := e)

block =
  do
    symbol "{"
    ps <- many program
    symbol "}"
    return (Block ps)

whileStatement =
  do
    symbol "while"
    symbol "("
    e <- expr
    symbol ")"
    p <- program
    return (While e p)

ifStatement =
```

```
  do
    symbol "if"
    symbol "("
    e <- expr
    symbol ")"
    p1 <- program
    ((do
        symbol "else"
        p2 <- program
        return (IfElse e p1 p2))
      <|>
       (return (If e p1)))

--------------------------------------------------------------------------------
---------------- DO **NOT** MAKE ANY CHANGES ABOVE THIS LINE --------------------
--------------------------------------------------------------------------------

--          |


readStatement = do
                    symbol "read"
                    val <- ident
                    symbol ";"
                    return (Read val)

writeStatement = do
                    symbol "write"
                    e <- expr
                    symbol ";"
                    return (Write e)

-- printStatement = do
--                    symbol "print"
--                    symbol "\""
--                    space
--                    s <- many (alphanum
--                                  <|>
--                                  (do
--                                      space
--                                      alphanum
--                                      ))



--                    symbol "\""
--                    symbol ";"
--                    return (Print s)

notSpeech :: Parser Char
notSpeech = sat (\inp -> if (inp == '\"') then False else True)

printStatement = do
                    symbol "print"
                    symbol "\""
                    space
                    s <- many (notSpeech)
                    symbol "\""
                    symbol ";"
```

```
                    return (Print s)

forStatement = do
    symbol "for"
    symbol "("
    i <- ident
    symbol "<-"
    mn <- expr
    symbol ".."
    mx <- expr
    symbol ")"
    p <- program
    return (For i mn mx p)

--------------------------------------------------------------------------------
---------------- DO **NOT** MAKE ANY CHANGES BELOW THIS LINE --------------------
--------------------------------------------------------------------------------

binExpr :: Parser e -> Parser ([e] -> e) -> Parser e -> Parser e
binExpr expr' op expr =
  do
    e' <- expr'
    ((do
        o <- op
        e <- expr
        return (o [e',e]))
      <|>
        return e')

expr  = binExpr expr1 orOp   expr
expr1 = binExpr expr2 andOp  expr1
expr2 = binExpr expr3 eqOp   expr2
expr3 = binExpr expr4 compOp expr3
expr4 = binExpr expr5 addOp  expr4
expr5 = binExpr expr6 mulOp  expr5

expr6 = expr7
     <|>
        do
          op <- notOp
          e <- expr6
          return (op [e])

expr7 = constant
    <|> do
          i <- identif
          return (Var i)
    <|> do
          symbol "("
          e <- expr
          symbol ")"
          return e

parseOp :: String -> OpName -> Parser ([Expr] -> Expr)
parseOp s op = do
                 symbol s
                 return (Op op)

orOp   = parseOp "||" Or
```

```
andOp  = parseOp "&&" And

eqOp   = parseOp "==" Eq

compOp = parseOp "<=" Leq
     <|> parseOp "<"  Less
     <|> parseOp ">=" Geq
     <|> parseOp ">"  Greater

addOp  = parseOp "+"  Add
     <|> parseOp "-"  Sub

mulOp  = parseOp "*"  Mul
     <|> parseOp "/"  Div
     <|> parseOp "%"  Mod

notOp  = parseOp "!"  Not

constant :: Parser Expr
constant = do
             n <- integer
             return (Constant(toInteger n))

keywords :: [String]
keywords = ["if", "else", "while", "for", "read", "write", "print"]


identif :: Parser String
identif =
  do
   cs <- token identifier
   guard (not (elem cs keywords))
   return cs

parseProgram :: String -> Program
parseProgram xs = case parse program xs of
                   [(p , [])] -> p
                   [(_ , s)]  -> error ("syntax: unparsed string " ++ s)
                   _          -> error "syntax: failed to parse program"
```