phk851 & hlp179 - Advanced Programming 3
September 30, 2022

Introduction This assignment will describe the design choices of our implementation of a Boa parser. If implemented correct it should convert a well written program to an Abstract Syntax Tree.

Design choices and implementation

Grammar: From the assignment we were handed out grammar for a BoaParser a . In the generical lexical convension especially the grammar for *Expr* is highly amibguous. The grammar does not take into consideration the precedence of different operators. This means that some operators are grouped looser than others. For instance the original grammar of *Expr* does not handle the precedence that * are to be evaluated prior to +, which in turn should be evaluated prior '=='. This is something that can be handled by modifying the grammar of *Expr*. The original grammar is also left-recursive, which means that there is a derivation of $Expr \implies Expr$. As we are doing top down parsing we will be unable to evaluate the alternatives for *Expr* by looking at the first input. An additional consideration is that the loosest grouping is the logical-negation `'not'`, which also can be nested. Meaning that you can write `'not' 'not' 'not'` which evaluates to `not (not (not ))` .

In order to grasp these characteristics we have modified the grammar of *Expr*. This is done using the technique for changing grammar such that is not left recursive from *parsernotes.pdf* section 3.4. If we have a nonterminal `A` and a sequence of one or more `g`'s we can introduce a new nonterminal e.g. Aopt which can derive zero or more `g`'s and we will be able to reconstruct the same string as the original grammar would produce.
To handle recurring `not` operations, this is evaluated at the beginning if there is a `'not'` term followed by a tighter expression and then if there are preceding `'not'`. The grammar can be seen below. We follow this patterns by creating sub expressions which are to be evaluated before other operations. This ensures that e.g. `x*y+z == x-y//s` will be evaluated as `((x*y)+z) == (x-(y/s))`. This can be seen in the grammar where if there were a factor it would be evaluated in `FactorOpt` where as the rest would be evaluated in `TermOpt` or `ExprOpt` which will handle the tightest and loosest relation accordingly.
Finally all of our terminals and nonterminals with zero or more occurrences are evaluated in X, such as *numConst*, *stringConst* and such.

```
     Expr ::= ExprNot
  ExprNot ::= 'not' N ExprNot | ϵ
        N ::= Term ExprOpt
  ExprOpt ::= '==' Term | '!=' Term | '<' Term | '<=' Term | '>' Term
            | '>=' Term | 'in' Term | 'not' 'in' Term
            | ϵ
     Term ::= Factor TermOpt
  TermOpt ::= '+' Factor TermOpt | '-' Factor TermOpt | ϵ
   Factor ::= X FactorOpt
FactorOpt ::= '*' X FactorOpt | '//' X FactorOpt | '%' X FactorOpt
            | ϵ
        X ::= numConst | stringConst | ident | '(' Expr ')'
            | ident '(' Expr ')' | '[' Exprz ']' |
            | '[' Expr ForClause Clausez ']' | 'None' | 'True' | 'False'
```

Assessment

Completeness All the intended functionality is implemented, while not all working correctly as desired however. This can be seen through the provided tests, where some successful tests can be found at every branch. From running `stack test --coverage` the test have a 100% coverage of booleans, if-conditions and qualifiers, 95% expression used and 70% top-level declaritions used.

Correctness The parser was thoroughly tested through tasty and by evaluating the test results, most of the functionality works as intended. Currently the parser can't handle input strings containing backslashes with newline, tab or escapes like: `"This is a \n string"`, `"This is \t a string"`, `"This is \\a string"`, `"This is \\\' a string"`. This is also the issue for evaluating *stringConst*, where it is unable to escape for `', \` and `\n` properly. The issue with the strings could likely be solved by changing our grammar such that a string is a string potentially followed by a \followed by a string.

Our implementation fails when it tries to parse the string "`xfor x in y`". This is parsed as a correct for-clause where it should fail, which must be due to the parsers ability to handle white space. Additionally the string "- 1" shouldn't parse, however it does in this implementation which might be due to an extensive use of `lexeme` which removes whitespace.

Efficiency The efficiency of the code is for the majority as one could expect of a parser. However deep nested brackets is handled too slow which could potentially use a lot of memory. This could be due to the indirect recursion in the grammar from "'[' *Exprz* ']'" → "*Exprs*" → *Expr* → "'[' *Exprz* ']'".

Maintainability Most of tests look identical where few parameters was changed which could be packed into a function, at least for the failed tests. Not enough time was used for commenting the code, however it follows to a large extent the grammar, and thus most of the interesting functions describes how it should handle each terminal/nonterminal.

Concluding remarks The parser are built up such that it is able to handle some of the major lexical conventions of Boa, however there are issues in parsing programs containing some special cases where the syntax does not explicitly meet the Boa syntax. Parsing *stringConst* are unable to escape, therefore only 'plain' strings are able to be parsed.

Appendix

BoaParser.hs

```
-- Skeleton file for Boa Parser.

module BoaParser (ParseError, parseString) where
import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))
import Data.Char
import BoaAST
import Control.Monad
-- add any other other imports you need

type Parser a = ReadP a

type ParseError = String -- you may replace this
reserved :: [String]
reserved = ["None", "True", "False", "for", "if", "in", "not"]

keyword :: String -> Parser ()
keyword s = lexeme $ do s' <- munch isAlphaNum
                        Control.Monad.unless (s' == s) pfail

pProgram :: Parser [Stmt]
pProgram = do pStmts

pStmts :: Parser [Stmt]
pStmts = do sm <- pStmt; return [sm]
            <|> do sm <- pStmt
                   symbol ";"
                   sms <- pStmts
                   return (sm: sms)


pStmt :: Parser Stmt
pStmt = do var <- pIdent; symbol "="; e <- pExpr; return $ SDef var e
       <|> do e <- pExpr; return $ SExp e

pExpr :: Parser Exp
pExpr = do pExprNot

pExprNot :: Parser Exp
pExprNot = do keyword "not"; Not <$> pExpr
          <|> do t1 <- pTerm; pExprOpt t1

pExprOpt :: Exp -> Parser Exp
pExprOpt t1 = do eo <- getExprOpt; t2 <- pTerm; return (eo t1 t2)
              <|> return t1
```

```haskell
getExprOpt :: Parser (Exp -> Exp -> Exp)
getExprOpt = do symbol "=="; return $ Oper Eq
            <|> do symbol "<"; return $ Oper Less
            <|> do symbol ">"; return $ Oper Greater
            <|> do keyword "in"; return $ Oper In
            <|> do symbol "<="; return (\e1 e2 -> Not $ Oper Greater e1 e2)
            <|> do symbol ">="; return (\e1 e2 -> Not $ Oper Less e1 e2)
            <|> do symbol "!="; return (\e1 e2 -> Not $ Oper Eq e1 e2)
            <|> do keyword "not"; do keyword "in"
                   return (\e1 e2 -> Not $ Oper In e1 e2)

pTerm :: Parser Exp
pTerm = do f1 <- pFactor; pTermOpt f1

pTermOpt :: Exp -> Parser Exp
pTermOpt f1 = do to <- getTermOpt; f2 <- pFactor; pTermOpt (to f1 f2)
              <|> return f1

getTermOpt :: Parser (Exp -> Exp -> Exp)
getTermOpt = do symbol "+"; return $ Oper Plus
            <|> do symbol "-"; return $ Oper Minus

pFactor :: Parser Exp
pFactor = do x1 <- pX; pFactorOpt x1

pFactorOpt :: Exp -> Parser Exp
pFactorOpt x1 = do fo <- getFactorOpt; x2 <- pX; pFactorOpt (fo x1 x2)
              <|> return x1

getFactorOpt :: Parser (Exp -> Exp -> Exp)
getFactorOpt = do symbol "*"; return $ Oper Times
            <|> do symbol "//"; return $ Oper Div
            <|> do symbol "%"; return $ Oper Mod


pX :: Parser Exp
pX = lexeme (do n <- pNum; return $ Const $ IntVal n)
    <|> lexeme (do symbol "\'"; do s <- pString; symbol "\'"
                                   return $ Const $ StringVal s)
    <|> lexeme (do pAtom)
    <|> lexeme (do i <- pIdent; return $ Var i)
    <|> do i <- pIdent; symbol "("; ez <- pExprz; symbol ")"
           return $ Call i ez
    <|> do symbol "("; e <- pExpr; symbol ")"; return e
    <|> do symbol "["; e <- pExpr; c <- pCCFor; cs <- pClausez; symbol "]"
           return $ Compr e (c:cs)
    <|> do symbol "[";ez <- pExprz; symbol "]"; return $ List ez
```

```haskell
pAtom :: Parser Exp
pAtom = do keyword "True"; return $ Const TrueVal
        <|> do keyword "False"; return $ Const FalseVal
        <|> do keyword "None"; return $ Const NoneVal


pIdent :: Parser String
pIdent = lexeme $ do c <- satisfy(\c -> isAlpha c || c == '_')
                     cs <- many (satisfy (\c -> isAlphaNum c || c == '_'))
                     let i = c:cs
                     if i `notElem` reserved then return i
                     else do pfail


pStringAgain :: Parser String
pStringAgain = do symbol "\n";s <- pString; return $ "\n" ++ s
         <|> do symbol "\\"; s <- pString; return $ "\\" ++ s
         <|> do symbol "\'"; s <- pString; return $ "'" ++ s


pString :: Parser String
pString = do s <- many(satisfy isAscii)
             x <- many pStringAgain
             return $ s ++ concat x


pNum :: Parser Int
pNum = do _ <- char '0'
          many1 (satisfy isDigit)
          pfail
      <|> lexeme (do _ <- symbol "-" -- handles negative numbers
                     d <- satisfy (\char -> char >= '1' && char <= '9')
                     ds <- many (satisfy isDigit)
                     return $ negate $ read $ d : ds)
      <|> lexeme (do d <- (satisfy (\char -> char >= '1' && char <= '9'))
                     ds <- many (satisfy isDigit)
                     return $ read $ d : ds)
      <|> lexeme (do _ <- skipMany (satisfy (== '-')); d <- string "0"
                     return $ read d)


pClausez :: Parser [CClause]
pClausez = do return []
          <|> do ccFor <- pCCFor; cs <- pClausez; return (ccFor : cs)
          <|> do ccIf <- pCCIf; cs <- pClausez; return (ccIf : cs)
```

```haskell
pCCFor :: Parser CClause
pCCFor = lexeme $ do _ <- keyword "for"
                     var <- pIdent
                     _ <- keyword "in"
                     e <- pExpr
                     return $ CCFor var e

pCCIf :: Parser CClause
pCCIf = lexeme $ do _ <- keyword "if"
                    e <- pExpr
                    return $ CCIf e


pExprz :: Parser [Exp]
pExprz = do return []
            <|> do pExprs

-- Containing 1 or more Exps.
pExprs :: Parser [Exp]
pExprs = do e <- pExpr
            symbol ","
            es <- pExprs
            return (e:es)
            <|> do e1 <- pExpr; return [e1]

--------------HELPER FUNCTIONS--------------
whitespace :: Parser ()
whitespace =
    do _ <- skipSpaces; return ()

lexeme :: Parser a -> Parser a
lexeme p = do a <- p; whitespace; return a

symbol :: String -> Parser ()
symbol s = lexeme $ do string s; return ()

parseString :: String -> Either ParseError Program
parseString s = case readP_to_S (do whitespace; p <- pProgram; eof;
                                    return p) s of
                    [] -> Left "Cannot parse"
                    [(p, _)] -> Right p
                    _ -> error "Grammar is ambiguous!"
```

Test.hs

```
-- Rudimentary test suite. Feel free to replace anything.

import BoaAST
import BoaParser

import Test.Tasty
import Test.Tasty.HUnit

main :: IO ()
main = defaultMain $ atomTests

atomTests = testGroup "Minimal pAtom" [
-------------------------------------------------------------------------pBool---------------------
  testCase "pBool1" $ parseString "True"
    @?= Right [SExp (Const (TrueVal))],
  testCase "pBool2" $ parseString "False"
    @?= Right [SExp (Const (FalseVal))],
  testCase "pBool3" $ parseString "None"
    @?= Right [SExp (Const (NoneVal))],


-------------------------------------------------------------------------pNum----------------------
  testCase "pNum1" $ parseString "5"
    @?= Right [SExp (Const (IntVal 5))],
  testCase "pNum2" $ parseString "5593"
    @?= Right [SExp (Const (IntVal 5593))],
  testCase "pNum3" $ parseString "-5"

    @?= Right [SExp (Const (IntVal (-5)))],
  testCase "pNum4" $ parseString "-554328"
    @?= Right [SExp (Const (IntVal (-554328)))],
  testCase "pNum5" $ parseString "0"
    @?= Right [SExp (Const (IntVal 0))],
  testCase "pNum6" $ parseString "-0"
    @?= Right [SExp (Const (IntVal 0))],
  testCase "pNumFail1" $ case parseString "- 1" of
                           Left e -> return ()
                           Right p -> assertFailure $
                                      "Unexpected parse: "
                                      ++ show p,

  testCase "pNumFail2" $ case parseString "01" of
                           Left e -> return ()
                           Right p -> assertFailure $
                                      "Unexpected parse: "
                                      ++ show p,
```

```
  testCase "pNumFail3" $ case parseString "-01" of
                           Left e -> return ()
                           Right p -> assertFailure $
                                      "Unexpected parse: "
                                      ++ show p,


-------------------------------------------------------------------------pIdent--------------------
  testCase "pIdent1" $ parseString "x"
    @?= Right [SExp (Var ("x"))],
  testCase "pIdent2" $ parseString "_x"
    @?= Right [SExp (Var ("_x"))],
  testCase "pIdent3" $ parseString "_1"
    @?= Right [SExp (Var ("_1"))],
  testCase "pIdent4" $ parseString "var"
    @?= Right [SExp (Var ("var"))],
  testCase "pIdent5" $ parseString "var1_"
    @?= Right [SExp (Var ("var1_"))],
  testCase "pIdent6" $ parseString "_var1_"
    @?= Right [SExp (Var ("_var1_"))],
  testCase "pIdent7" $ parseString "_4var1_"
    @?= Right [SExp (Var ("_4var1_"))],
  testCase "pIdent8" $ parseString "_4var1_"
    @?= Right [SExp (Var ("_4var1_"))],


-------------------------------------------------------------------pIdentFail----------------
  testCase "pIdent1" $ case parseString "2var1_" of
                           Left e -> return ()
                           Right p -> assertFailure $
                                      "Unexpected parse: "
                                      ++ show p,
  testCase "pIdent2" $ case parseString "3_var1_" of
                           Left e -> return ()
                           Right p -> assertFailure $
                                      "Unexpected parse: "
                                      ++ show p,
  testCase "pIdent3" $ case parseString "1x" of
                           Left e -> return ()
                           Right p -> assertFailure $
                                      "Unexpected parse: "
                                      ++ show p,
  testCase "pIdent4" $ case parseString "for" of
                           Left e -> return ()
                           Right p -> assertFailure $
                                      "Unexpected parse: "
                                      ++ show p,
  testCase "pIdent5" $ case parseString "if" of
```

```
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent6" $ case parseString "in" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent7" $ case parseString "not" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent8" $ case parseString "1_" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent9" $ case parseString "!a" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent10" $ case parseString "a\"" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent11" $ case parseString "%a" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent12" $ case parseString "&a" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent13" $ case parseString "/a" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
                                   ++ show p,
    testCase "pIdent14" $ case parseString "(a" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                   "Unexpected parse: "
```

```
                                    ++ show p,
  testCase "pIdent15" $ case parseString ")a" of
                          Left e -> return ()
                          Right p -> assertFailure $
                                     "Unexpected parse: "
                                     ++ show p,
```

---pExprOpt---

```
  testCase "ExprOpt1" $ parseString "x < 5"
    @?= Right [SExp (Oper Less (Var "x") (Const (IntVal 5)))],
  testCase "ExprOpt2" $ parseString "x > 5"
    @?= Right [SExp (Oper Greater (Var "x") (Const (IntVal 5)))],
  testCase "ExprOpt3" $ parseString "x == 5"
    @?= Right [SExp (Oper Eq (Var "x") (Const (IntVal 5)))],
  testCase "ExprOpt4" $ parseString "x >= 5"
    @?= Right [SExp (Not (Oper Less (Var "x") (Const (IntVal 5))))],
  testCase "ExprOpt5" $ parseString "x <= 5"
    @?= Right [SExp (Not (Oper Greater (Var "x") (Const (IntVal 5))))],
  testCase "ExprOpt6" $ parseString "x != 5"
    @?= Right [SExp (Not (Oper Eq (Var "x") (Const (IntVal 5))))],
  testCase "ExprOpt7" $ parseString "x in 5"
    @?= Right [SExp (Oper In (Var "x") (Const (IntVal 5)))],
  testCase "ExprOpt8" $ parseString "True == False"
    @?= Right [SExp (Oper Eq (Const (TrueVal)) (Const (FalseVal)))],
  testCase "ExprOpt9" $ parseString "x not in 5"
    @?= Right [SExp (Not (Oper In (Var "x") (Const (IntVal 5))))],
  testCase "ExprOpt9" $ parseString "True != False"
    @?= Right [SExp (Not (Oper Eq (Const (TrueVal)) (Const (FalseVal))))],


  testCase "ExprOptFail1" $
    case parseString "1 < 2 < 3" of
    Left e -> return ()
    Right p -> assertFailure $ "Unexpected parse: " ++ show p,
  testCase "ExprOptFail2" $
    case parseString "1 > 2 > 3" of
    Left e -> return ()
    Right p -> assertFailure $ "Unexpected parse: " ++ show p,
  testCase "ExprOptFail3" $
    case parseString "1 == 2 == 3" of
    Left e -> return ()
    Right p -> assertFailure $ "Unexpected parse: " ++ show p,
  testCase "ExprOptFail4" $
    case parseString "1 >= 2 >= 3" of
    Left e -> return ()
    Right p -> assertFailure $ "Unexpected parse: " ++ show p,
```

```
  testCase "ExprOptFail5" $
    case parseString "1 <= 2 <= 3" of
    Left e -> return ()
    Right p -> assertFailure $ "Unexpected parse: " ++ show p,
  testCase "ExprOptFail6" $
    case parseString "1 in 2 in 3" of
    Left e -> return ()
    Right p -> assertFailure $ "Unexpected parse: " ++ show p,
  testCase "ExprOptFail7" $
    case parseString "1 != 2 != 3" of
    Left e -> return ()
    Right p -> assertFailure $ "Unexpected parse: " ++ show p,

----------------------------------------------------------------------Not----------------------

  testCase "NUUT NUUT" $ parseString "not not x + 5"
    @?= Right [SExp (Not (Not (Oper Plus (Var "x") (Const (IntVal 5)))))],
----------------------------------------------------------------------TermOpt------------------

  testCase "TermOpt1" $ parseString "x + 5"
    @?= Right [SExp (Oper Plus (Var "x") (Const (IntVal 5)))],
  testCase "TermOpt2" $ parseString "x - 5"
    @?= Right [SExp (Oper Minus (Var "x") (Const (IntVal 5)))],
  testCase "TermOpt3" $ parseString "x + y"
    @?= Right [SExp (Oper Plus (Var "x") (Var "y"))],
  testCase "TermOpt4" $ parseString "x - y"
    @?= Right [SExp (Oper Minus (Var "x") (Var "y"))],
  testCase "TermOpt5" $ parseString "0 - -5"
    @?= Right [SExp (Oper Minus (Const (IntVal 0)) (Const (IntVal (-5))))],
  testCase "TermOpt6" $ parseString "0 + -5"
    @?= Right [SExp (Oper Plus (Const (IntVal 0)) (Const (IntVal (-5))))],
  testCase "TermOpt7" $ parseString "-0 - -5"
    @?= Right [SExp (Oper Minus (Const (IntVal 0)) (Const (IntVal (-5))))],
  testCase "TermOptFail1" $ case parseString "00 - -5" of
                            Left e -> return ()
                            Right p -> assertFailure $
                                       "Unexpected parse: "
                                       ++ show p,
  testCase "TermOptFail2" $ case parseString "5 - -001" of
                            Left e -> return ()
                            Right p -> assertFailure $
                                       "Unexpected parse: "
                                       ++ show p,

----------------------------------------------------------------------FactorOpt----------------

  testCase "FactorOpt1" $ parseString "10 * -5"
    @?= Right [SExp (Oper Times (Const (IntVal 10)) (Const (IntVal (-5))))],
```

```
  testCase "FactorOpt2" $ parseString "-0 // -5"
    @?= Right [SExp (Oper Div (Const (IntVal 0)) (Const (IntVal (-5))))],
  testCase "FactorOpt3" $ parseString "20 // 3"
    @?= Right [SExp (Oper Div (Const (IntVal 20)) (Const (IntVal (3))))],
  testCase "FactorOpt4" $ parseString "20 // 0"
    @?= Right [SExp (Oper Div (Const (IntVal 20)) (Const (IntVal (0))))],
  testCase "FactorOpt5" $ parseString "20 % 40"
    @?= Right [SExp (Oper Mod (Const (IntVal 20)) (Const (IntVal (40))))],


-------------------------------------------------------------------------FactorTermExprOpt----------

  testCase "FactorOpt&TermOpt&ExprOpt1" $
    parseString "x + y < x - y"
    @?= Right [SExp (Oper Less (Oper Plus (Var "x") (Var "y"))
              (Oper Minus (Var "x") (Var "y")))],
  testCase "FactorOpt&TermOpt&ExprOpt2" $
    parseString "x * y > x // y"
    @?= Right [SExp (Oper Greater (Oper Times (Var "x") (Var "y"))
              (Oper Div (Var "x") (Var "y")))],
  testCase "FactorOpt&TermOpt&ExprOpt3" $
  parseString "x + y <= x // y"
  @?= Right [SExp (Not (Oper Greater (Oper Plus (Var "x") (Var "y"))
            (Oper Div (Var "x") (Var "y"))))],
  testCase "FactorOpt&TermOpt&ExprOpt4" $
  parseString "x * y >= x - y"
  @?= Right [SExp (Not (Oper Less (Oper Times (Var "x") (Var "y"))
            (Oper Minus (Var "x") (Var "y"))))],


-------------------------------------------------------------------------pString--------------------

  testCase "pString1" $
    parseString "'This is a string'"
    @?= Right [SExp (Const (StringVal ("This is a string")))],
  testCase "pString2" $
    parseString "'This is a \n string'"
    @?= Right [SExp (Const (StringVal ("This is a string")))],
  testCase "pString3" $
    parseString "'This is \t a string'"
    @?= Right [SExp (Const (StringVal ("This is \t a string")))],
  testCase "pString4" $
    parseString "'This is \\ a string'"
    @?= Right [SExp (Const (StringVal ("This is \\ a string")))],
  testCase "pString5" $
    parseString "'This is \\\' a string'"
    @?= Right [SExp (Const (StringVal ("This is \' a string")))],
  testCase "pString6" $
```

```
  parseString "'This is a com#ment'"
  @?= Right [SExp (Const (StringVal ("This is a com")))],
testCase "pString7" $
  parseString "''"
  @?= Right [SExp (Const (StringVal (""))),

testCase "pStringFail1" $ case parseString "'This is no#t a string" of
                        Left e -> return ()
                        Right p -> assertFailure
                                    $ "Unexpected parse: "
                                    ++ show p,
testCase "pStringFail2" $ case parseString "'This is no#t a string" of
                        Left e -> return ()
                        Right p -> assertFailure
                                    $ "Unexpected parse: "
                                    ++ show p,
testCase "pStringFail3" $ case parseString "'This is not \\\a string'" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                    "Unexpected parse: "
                                    ++ show p,
testCase "pStringFail4" $ case parseString "'This is not \a string'" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                    "Unexpected parse: "
                                    ++ show p,
testCase "pStringFail5" $ case parseString "'This is no§t a string" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                    "Unexpected parse: "
                                    ++ show p,
testCase "pStringFail6" $ case parseString "'This is no¾t a string" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                    "Unexpected parse: "
                                    ++ show p,
testCase "pStringFail7" $ case parseString "'This is no½t a string" of
                        Left e -> return ()
                        Right p -> assertFailure $
                                    "Unexpected parse: "
                                    ++ show p,

-------------------------------------------------------------------Compr---------------------

testCase "Compr1" $
  parseString "[x for x in x]"
  @?= Right [SExp (Compr (Var ("x")) [CCFor "x" (Var "x")])],
testCase "Compr5" $
```

```
  parseString "[x for x in x if u]"
    @?= Right [SExp (Compr (Var ("x")) [CCFor "x" (Var "x"), CCIf (Var "u")])],
testCase "Compr6" $
  parseString "[x for x in x in u]"
  @?= Right [SExp (Compr (Var "x") [CCFor "x" (Oper In (Var "x") (Var "u"))])],
testCase "Compr7" $
  parseString "[x+2 for x in x for y in 7 if 1 < 2]"
    @?= Right [SExp (Compr (Oper Plus (Var "x") (Const (IntVal 2)))
              [CCFor "x" (Var "x"),CCFor "y" (Const (IntVal 7)),
              CCIf (Oper Less (Const (IntVal 1)) (Const (IntVal 2)))])],
testCase "ComprFail1" $ case parseString "[x for x in x ift]" of
                        Left e -> return ()
                        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
testCase "ComprFail2" $ case parseString "[xfor y in z]" of
                        Left e -> return ()
                        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
testCase "ComprFail3" $ case parseString "[x forz in k]" of
                        Left e -> return ()
                        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
```

---------- *pExprs* ----------

```
testCase "pExprs1" $
  parseString "5; 4"
  @?= Right [SExp (Const (IntVal 5)), SExp (Const (IntVal 4))],
testCase "pExprs2" $
  parseString "5 + x; 4"
  @?= Right [SExp (Oper Plus (Const (IntVal 5)) (Var "x")),
            SExp (Const (IntVal 4))],
testCase "pExprs3" $
  parseString "'This is a string'; 4"
  @?= Right [SExp (Const (StringVal "This is a string")),
            SExp (Const (IntVal 4))],
testCase "pExprs4" $
  parseString "'This is a string'; 4; [x for x in x]"
  @?= Right [SExp (Const (StringVal "This is a string")),
            SExp (Const (IntVal 4)), SExp (Compr (Var ("x"))
            [CCFor "x" (Var "x")])],
testCase "pExprs5" $
  parseString "'This is a string'; 4 % 8; [x for x in x]"
  @?= Right [SExp (Const (StringVal "This is a string")),
            SExp (Oper Mod (Const (IntVal 4)) (Const (IntVal 8))),
            SExp (Compr (Var "x") [CCFor "x" (Var "x")])],
testCase "pExprs6" $
  parseString "x=5; x+2"
  @?= Right [SDef "x" (Const (IntVal 5)),
            SExp (Oper Plus (Var "x") (Const (IntVal 2)))],
```

```
  testCase "pExprs7" $
    parseString "x=5; y=7; x+y"
    @?= Right [SDef "x" (Const (IntVal 5)),SDef "y" (Const (IntVal 7)),
               SExp (Oper Plus (Var "x") (Var "y"))],
  testCase "pExprs8" $
    parseString "x=5; y=7; [x for x in y if 1 < 2]"
      @?= Right [SDef "x" (Const (IntVal 5)),SDef "y" (Const (IntVal 7)),
                 SExp (Compr (Var "x") [CCFor "x" (Var "y"),
                 CCIf (Oper Less (Const (IntVal 1)) (Const (IntVal 2)))])],
  testCase "pExprs9" $
    parseString "x = 5; y = 7; [x for x in y if 1 < 2]"
      @?= Right [SDef "x" (Const (IntVal 5)),SDef "y" (Const (IntVal 7)),
                 SExp (Compr (Var "x") [CCFor "x" (Var "y"),
                 CCIf (Oper Less (Const (IntVal 1)) (Const (IntVal 2)))])],
  testCase "pExprs10" $
    parseString "x = 5; y = 7*x; [x for x in y if 1 < 2]"
      @?= Right [SDef "x" (Const (IntVal 5)),SDef "y"
                 (Oper Times (Const (IntVal 7)) (Var "x")),
                 SExp (Compr (Var "x") [CCFor "x" (Var "y"),CCIf
                 (Oper Less (Const (IntVal 1)) (Const (IntVal 2)))])],
  testCase "pExprs11" $
    parseString "x = 5; y = 7; [x      for x    in y     if 1  < 2    ]"
      @?= Right [SDef "x" (Const (IntVal 5)),SDef "y"
                 (Const (IntVal 7)),SExp (Compr (Var "x")
                 [CCFor "x" (Var "y"),CCIf (Oper Less (Const (IntVal 1))
                 (Const (IntVal 2)))])],


-------------------------------------------------------------------------pIdent pExprs--------------
  testCase "ident (Exp)" $ parseString "range (5)"
    @?= Right [SExp (Call "range" [Const (IntVal 5)])],
  testCase "ident (Exp)" $ parseString "range (5+y)"
    @?= Right [SExp (Call "range"
    [Oper Plus (Const (IntVal 5)) (Var "y")])],
  testCase "ident (Exp)" $ parseString "print (5)"
    @?= Right [SExp (Call "print" [Const (IntVal 5)])],
  testCase "ident (Exp)" $ parseString "print (5 + 6)"
    @?= Right [SExp (Call "print"
    [Oper Plus (Const (IntVal 5)) (Const (IntVal 6))])],
  testCase "ident (Exp)" $ parseString "print ('This is a string')"
    @?= Right [SExp (Call "print" [Const (StringVal "This is a string")])],
  testCase "ident (Exp)" $ parseString "print (x == Y)"
    @?= Right [SExp (Call "print" [Oper Eq (Var "x") (Var "Y")])],
  testCase "finaltest" $ parseString "1"
    @?= Right [SExp (Const (IntVal 1))]
  ]
```