Advanced Programming

Assignment 3

University of Copenhagen

Julian Sonne Westh Wulff <bkx655@alumni.ku.dk>
Toke Emil Heldbo Reines <bkx591@alumni.ku.dk>

September 29, 2021

1 Completeness

In this assignment, we implemented a parser for the Boa language that translates the concrete syntax of the Boa language into the abstract syntax used for the Boa interpreter from the last assignment. To implement the parser we choose to use Parsec.

Our implementation of the parser is currently implemented to parse and the concrete syntax of Boa, its disambiguation, and the defined correspondence between the concrete and abstract syntax. Thus the parser implements all asked-for functionality. Furthermore, we also included a test suite for doing automated tests of parsing the concrete Boa language into the abstract syntax within the specifications given in the assignment. To run the automated test suite navigate to the `code/part2/` folder and run the `stack test` command to build and run the tests.

In the following subsections, we will shortly describe some of the non-trivial design and implementations choices we have made, including modifications to the grammar to making it suitable for the Parsec parser library.

1.1 Changes to the grammar

Firstly we choose to divide our main `Exp` parser into 3 parser i.e. `pExp`, `pExp'`, and `pExp''`. This was done in order to deal with the disambiguation of not allowing chaining of the relational operators as they were non-associative, forcing left-associativity of the additive and multiplicative arithmetic operators, and forcing the correct precedence's levels. Thus in `pExp` we simply start off by parsing non-chained relational operations, and if that fails we parse `pExp'`. In `pExp'` we then start of by parsing arithmetic operators using a two-leveled `chainl1` parser (`pOper`) that takes care of the precedence levels of the arithmetic operators and left recursion problem, as shown inListing 1. If the `pOper` doesn't parse, we then try to parse `pExp''`, which is the parser that parses the rest of the different types of expressions.

```
pOper :: Parser Exp
pOper = pOper' `chainl1` pAddOp

pOper' :: Parser Exp
pOper' = pExp'' `chainl1` pMulOp
```

Listing 1: Parsing arithmetic operations with left associativity and correct precedence

Besides the above-mentioned changes to the grammar mentioned above, we also had to change the order of some of the parsers compared to the order of the given Concrete syntax of Boa. This includes parsing *ident* '(' *Exprz* ')' before the standalone *ident*. As the parser would otherwise always parse the standalone *ident* first. We also choose to parse '[' *Expr ForClause Clausez* ']' before '[' *Exprz*']' as we had problems making the parser work without this modification as the parser would succeed with parsing '[' *Exprz* until failing on a *ForClause*. Tough this behavior could be mitigated by just wrapping '[' *Exprz*']' in a try and keeping the original order.

1.2 Backtracking/lookahead using try

In order to make the parser work correctly, we had to introduce backtracking/lookahead by using `try` several places. The main reason for this was to mitigate failing parser in consuming any tokens on failure. Thus we were forced to use `try` anywhere where we had to parse more than one char to check if the result had some specific structure, such as for identifiers, strings, or the operators consisting of several chars. Furthermore, we also had to introduce `try` to all but the last alternatives in the parsers for *Stmts*, *Stmt*, and *Expr* to eliminate the consummation of tokens further down the parser tree if they failed.

2 Correctness

All of the given concrete grammar of Boa has been thoroughly tested with our own test suite and with the online TA with no reported errors. For this reason feel fairly sure that the code is correct. But we can't guaranty this, as we have not made any correctness proofs, and their might be some edge cases that we have missed.

3 Efficiency

Due to the "extensive" use of `try` throughout the code, we believe that our code might in some cases be less efficient than it could be. As it makes the parser do a lot of unnecessary backtracking. Some of the inefficiency could probably be mitigated by a more clever grammar and doing more extensive tests trying to remove unnecessary `try`'s as some of the used `try`'s might be redundant.

4 Robustness

To the best of our knowledge, the code should be somewhat robust and shouldn't throw any runtime errors causing the program to crash and rather returns somewhat descriptive error messages where possible within the scope of the assignment.

We would have liked to dig into returning useful error-messages, as we are sure that that would give us even more insight in our parsers. But due to limited time and out-of-scope of this task, we did not implement this.

5 Maintainability

Our code is somewhat maintainable and structured such that the "main" parsers in the code follow the flow of the given concrete Boa syntax. Beneath the main parsers we made sections for our "helper parsers" (such as comments parsers, whitespace/lexeme parsers, and operator parsers), "satisfy helpers" (custom `Char -> Bool` functions), and lastly constants for reserved keywords, etc. Thus we have made sure to use parameterized auxiliary definitions to reduce code duplication, reduce the cognitive load of individual functions, and keeping a good coding practice.

The code is commented where we deemed that the code did not speak for itself, or where "magic numbers" occur such as in `isBoaAlpha`.

Due to a lack of time and the scope of the assignment, our code could be a bit more maintainable and debuggable by implementing better error handling/descriptions, but as this was outside the scope of the assignment we did not do so. Furthermore, we also experienced some issues regarding correct parsing of whitespace as where demanded is in some cases and in other cases not. This forced us to use a mix of `lexeme` and `spaces` scattered around the code to make the parser fully functional. This might make the code a bit unmaintainable and could handle a bit more cleverly given more time.

6 Other

Our comment-handling is not state-of-the art either. We've gotten smarter while working with the task, but haven't settled completely on which symbols comments should terminate on, as well as which symbols should be allowed as comments - preferably even emojis or unicode? But for simplicity we chose to only allow characters where `isAscii c == True`

After having implemented part 2 of the assignment we realize that we could have done part 1 a bit more concise/clever by using `chainl1` and changing the grammar a bit from our original solution, but we did not have time to do this, but rather put it as a comment in that code.

7 Appendix

7.1 Code - WarmupReadP.hs

```haskell
module WarmupReadP where

-- Original grammar (E is start symbol):
--   E ::= E "+" T | E "-" T | T | "-" T .
--   T ::= num | "(" E ")" .
-- Lexical specifications:
--   num is one or more decimal digits (0-9)
--   tokens may be separated by arbtrary whitespace (spaces, tabs, newlines).

-- Rewritten grammar, without left-recursion:
-- E ::= T E' | "-" T E'
-- E' :== "+" T E' | "-" T E' | E
-- T ::= num | "(" E ")"

import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))
import Data.Char
  -- may use instead of +++ for easier portability to Parsec

type Parser a = ReadP a   -- may use synomym for easier portability to Parsec

type ParseError = String  -- not particularly informative with ReadP

data Exp = Num Int | Negate Exp | Add Exp Exp
  deriving (Eq, Show)

parseString :: String -> Either ParseError Exp
parseString s =
  case readP_to_S(do whitespace; a <- pE; eof; return a) s of
    [] -> Left "cannot parse"
    [(a,_)] -> Right a -- the_must be "", since 'eof' ok_-> error"oops, my grammar is ambiguous!
    _ -> error "You've made an oopsie"

-- E ::= T E' | "-" T E'
pE :: Parser Exp
pE = do e <- pT; pE' e;
     <|> do symbol '-'; e <- pT; pE' (Negate e);

-- E' :== "+" T E' | "-" T E' | E
pE' :: Exp -> Parser Exp
pE' e1 =  do ao <- pAddOp; e2 <- pT; pE' (ao e1 e2)
          <|>
          do no <- pNegOp; e2 <- pT; pE' (Add e1 (no e2))
          <|> return e1

-- T ::= num | "(" E ")"
```

```
pT :: Parser Exp
pT = do pNum;
     <|>
     do symbol '('; e <- pE; symbol ')'; return e


pNum :: Parser Exp
pNum =  lexeme $ do ds <- munch1 isDigit; return $ Num (read ds)

pAddOp :: Parser (Exp -> Exp -> Exp)
pAddOp = lexeme $ do symbol '+'; return Add

pNegOp :: Parser (Exp -> Exp)
pNegOp = lexeme $ do symbol '-'; return Negate

symbol :: Char -> Parser ()
symbol s = lexeme $ do satisfy(s ==); return ()

whitespace :: Parser ()
whitespace = do munch isSpace; return ()

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

resultOfString = Right (Add (Add (Negate (Num 1)) (Num 23)) (Negate (Negate (Num 456))))
testParseString =  parseString "-1+23-(-456)" == resultOfString
                && parseString "-1" == Right (Negate (Num 1))
                && parseString "1" == Right (Num 1)
                && parseString "1+2" == Right (Add (Num 1 ) (Num 2))
                && parseString "-1+2" == Right (Add (Negate (Num 1 )) (Num 2))
                && parseString "1-2" == Right (Add (Num 1) (Negate (Num 2)))
                && parseString "1-(-(-(-1)))" == Right (Add (Num 1) (Negate (Negate (Negate (Negate (Num 1))))))
                && parseString " - 1 + 23 - ( - 456 ) " == resultOfString
                && parseString "   -   1   +   23   -   (   -   456   ) " == resultOfString
                && parseString "  \
                \-   1\t  +\n   23   -   (   -   456   ) " == resultOfString
```

7.2 Code - WarmupParsec.hs

```haskell
module WarmupParsec where

-- Original grammar (E is start symbol):
--   E ::= E "+" T | E "-" T | T | "-" T .
--   T ::= num | "(" E ")" .
-- Lexical specifications:
--   num is one or more decimal digits (0-9)
--   tokens may be separated by arbtrary whitespace (spaces, tabs, newlines).

-- Lexical specifications:
--   num is one or more decimal digits (0-9)
--   tokens may be separated by arbtrary whitespace (spaces, tabs, newlines).

-- Rewritten grammar, without left-recursion:
-- E ::= "-" T E' | T E'
-- E' :== "+" T E' | "-" T E' | E
-- T ::= num | "(" E ")"

-- ! Properbly more correct?
-- Rewritten grammar, without left-recursion:
-- E ::= T E' | T
-- E' :== "+" T E' | "-" T E' | E
-- T ::= num | "(" E ")" | "-" E

import Text.ParserCombinators.Parsec -- exports a suitable type ParseError

data Exp = Num Int | Negate Exp | Add Exp Exp
  deriving (Eq, Show)

-- Optional: if not attempted, leave as undefined
parseString :: String -> Either ParseError Exp
parseString = parse (do spaces; a <- pE; eof; return a) ""

-- -- E ::= T E' | "-" T E'
pE :: Parser Exp
pE =
  do e <- pT; pE' e
    <|> do symbol '-'; e <- pT; pE' (Negate e)

-- -- E' :== "+" T E' | "-" T E' | E
pE' :: Exp -> Parser Exp
pE' e1 =
  do ao <- pAddOp; e2 <- pT; pE' (ao e1 e2)
    <|> do no <- pNegOp; e2 <- pT; pE' (Add e1 (no e2))
    <|> return e1

-- -- T ::= "(" E ")" | num
pT :: Parser Exp
pT =
```

```haskell
  do pNum
    <|> do symbol '('; e <- pE; symbol ')'; return e

pNum :: Parser Exp
pNum = lexeme $ do ds <- many1 digit; return $ Num (read ds :: Int)

pAddOp :: Parser (Exp -> Exp -> Exp)
pAddOp = lexeme $ do symbol '+'; return Add

pNegOp :: Parser (Exp -> Exp)
pNegOp = lexeme $ do symbol '-'; return Negate

symbol :: Char -> Parser ()
symbol s = lexeme $ do satisfy (s ==) <?> [s]; return ()

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

resultOfString = Right (Add (Add (Negate (Num 1)) (Num 23)) (Negate (Negate (Num 456))))

testParseString =
  parseString "-1+23-(-456)" == resultOfString
    && parseString "-1" == Right (Negate (Num 1))
    && parseString "1" == Right (Num 1)
    && parseString "1+2" == Right (Add (Num 1) (Num 2))
    && parseString "-1+2" == Right (Add (Negate (Num 1)) (Num 2))
    && parseString "1-2" == Right (Add (Num 1) (Negate (Num 2)))
    && parseString "1-(-(-(-1)))" == Right (Add (Num 1) (Negate (Negate (Negate (Negate (Num 1))))))
    && parseString " - 1 + 23 - ( - 456 ) " == resultOfString
    && parseString "   -   1   +   23   -   (   -   456   ) " == resultOfString
    && parseString
      "  \
      \-   1\t   +\n   23   -   (   -   456   ) "
      == resultOfString

-- ! consider negative tests?
```

7.3 Code - BoaParser.hs

```haskell
-- Skeleton file for Boa Parser.

module BoaParser (ParseError, parseString) where

import BoaAST
-- add any other other imports you need

import Data.Char
import Text.ParserCombinators.Parsec

parseString :: String -> Either ParseError Program
parseString = parse (do spaces; a <- pProgram; eof; return a) ""

-- Main Parsers

pProgram :: Parser Program
pProgram = do pStmts

pStmts :: Parser [Stmt]
pStmts =
  try (do stmt <- pStmt; symbol ';'; stmts <- pStmts; return (stmt : stmts))
    <|> do stmt <- pStmt; return [stmt]

pStmt :: Parser Stmt
pStmt =
  try (do i <- pIdent; spaces; symbol '='; SDef i <$> pExp)
    <|> do SExp <$> pExp

pExp :: Parser Exp
pExp =
  lexeme $
    pComments $
      try (do e1 <- pExp'; spaces; ro <- pRelNegOp; spaces; Not . ro e1 <$> pExp')
        <|> try (do e1 <- pExp'; spaces; ro <- pRelOp; spaces; ro e1 <$> pExp')
        <|> pExp'

pExp' :: Parser Exp
pExp' =
  do pOper
    <|> do pExp''

pExp'' :: Parser Exp
pExp'' =
  do pNum
    <|> do Const . StringVal <$> pStr
    <|> do Const <$> pNoneTrueFalse
    <|> try (do i <- pIdent; spaces; ez <- between (symbol '(') (symbol ')') pExpz; return $ Call i ez)
    <|> try (do Var <$> pIdent)
```

```haskell
    <|> try (do string "not"; spaces; Not <$> pExp)
    <|> do between (symbol '(') (symbol ')') pExp
    <|> try (do symbol '['; e <- pExp; spaces; fc <- pForC; spaces; cz <- pCz; symbol ']'; return $ Compr e (fc : cz))
    <|> do e <- between (symbol '[') (symbol ']') pExpz; return $ List e

-- Precedence level low
pOper :: Parser Exp
pOper = pOper' `chainl1` pAddOp

-- Precedence level medium
pOper' :: Parser Exp
pOper' = pExp'' `chainl1` pMulOp

-- ForClause
pForC :: Parser CClause
pForC = lexeme $ do try (string "for"); notFollowedBy (satisfy isBoaAlphaNum); spaces; i <- pIdent; spaces; string "in"; notFollowedBy (satisfy isBoaAlphaNum); spaces; CCFor i <$> pExp

-- IfClause
pIfC :: Parser CClause
pIfC = lexeme $ do try (string "if"); notFollowedBy (satisfy isBoaAlphaNum); spaces; CCIf <$> pExp

-- Clausez
pCz :: Parser [CClause]
pCz =
  lexeme $
    do f <- pForC; cz <- pCz; return (f : cz)
      <|> do i <- pIfC; cz <- pCz; return (i : cz)
      <|> return []

pExpz :: Parser [Exp]
pExpz = lexeme $ do pExp `sepBy` symbol ','

-- Not called anywhere in the code. Defined in the AST, which is why we kept it here as a commet.
-- pExps :: Parser [Exp]
-- pExps = lexeme £ do pExp `sepBy1` symbol ','

pIdent :: Parser String
pIdent =
  do
    c <- satisfy isIdentPrefixChar
    cs <- many (satisfy isIdentChar)
    let i = c : cs
    if i `notElem` reservedIdents
      then return i
      else fail "variable can't be a reserved word"
```

```haskell

numSign :: Parser Int
numSign =
  do satisfy (== '-'); return (-1)
    <|> return 1

pNum :: Parser Exp
pNum =
  lexeme $ do
    s <- numSign
    ds <- many1 digit
    case ds of
      [] -> fail ""
      [d] -> return $ Const (IntVal (read [d] * s))
      (d : ds') ->
        if d == '0'
          then fail "num constants cant start with 0"
          else return $ Const (IntVal (read (d : ds') * s))

pStr :: Parser String
pStr =
  try (do symbol' '\''; symbol '\''; return []) -- Checks for the empty string
    <|> try
      ( do
          symbol' '\''
          a <- concat <$> many1 escaped -- many1 makes sure to not return [] one chars such as \n \t
          if null a
            then do symbol' '\''; fail "string may only hold ASCII printable charecters"
            else do symbol' '\''; return a
      )
  where
    escaped =
      try (do string "\\\\"; return "\\")
        <|> try (do string "\\\'"; return "\'")
        <|> try (do string "\\\n"; return "")
        <|> try (do string "\\n"; return "\n")
        <|> do
          x <- noneOf ['\'', '\\', '\n']
          if isStrChar x
            then return [x]
            else fail "string may only hold ASCII printable charecters"

pNoneTrueFalse :: Parser Value
pNoneTrueFalse =
  lexeme $
    try (do string "None"; return NoneVal)
      <|> try (do string "True"; return TrueVal)
      <|> try (do string "False"; return FalseVal)

```

```haskell
-- Sub parsers

symbol :: Char -> Parser ()
symbol s = lexeme $ do satisfy (s ==); return ()

symbol' :: Char -> Parser ()
symbol' s = do satisfy (s ==); return ()

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

pComments' :: Parser ()
pComments' =
  try (do spaces; symbol' '#'; manyTill (satisfy isAscii) (char '\n'); return ())
    <|> try (do spaces; symbol' '#'; manyTill (satisfy isAscii) eof; return ()) -- ! Try could properly be removed

pComments :: Parser a -> Parser a
pComments p = do skipMany pComments'; a <- p; skipMany pComments'; return a

pRelOp :: Parser (Exp -> Exp -> Exp)
pRelOp =
  lexeme $
    do try (string "=="); return $ Oper Eq
      <|> do try (string "in"); notFollowedBy (satisfy isBoaAlphaNum); return $ Oper In
      <|> do Oper Less <$ symbol '<'
      <|> do Oper Greater <$ symbol '>'

pRelNegOp :: Parser (Exp -> Exp -> Exp)
pRelNegOp =
  lexeme $
    do try (string "!="); return $ Oper Eq
      <|> try (do string "not"; many1 $ satisfy isSpace; string "in"; return $ Oper In)
      <|> do try (string "<="); return $ Oper Greater
      <|> do try (string ">="); return $ Oper Less

pAddOp :: Parser (Exp -> Exp -> Exp)
pAddOp =
  lexeme $
    do symbol '+'; notFollowedBy isRel; return $ Oper Plus
      <|> do symbol '-'; notFollowedBy isRel; return $ Oper Minus

pMulOp :: Parser (Exp -> Exp -> Exp)
pMulOp =
  lexeme $
    do symbol '*'; notFollowedBy isRel; return $ Oper Times
      <|> try (do string "//"; notFollowedBy isRel; return $ Oper Div)
      <|> do symbol '%'; notFollowedBy isRel; return $ Oper Mod

isRel :: Parser ()
```

```haskell
isRel =
  lexeme $
    do symbol '<'; return ()
      <|> do symbol '>'; return ()
      <|> try (do string "=="; return ())
      <|> try (do string "in"; return ())
      <|> try (do string "<="; return ())
      <|> try (do string ">="; return ())
      <|> try (do string "!="; return ())
      <|> try (do string "not"; spaces; string "in"; return ())
      <|> try (do string "not"; return ())

-- Satisfy helpers

isIdentPrefixChar :: Char -> Bool
isIdentPrefixChar c = isBoaAlpha c || c == '_'

isIdentChar :: Char -> Bool
isIdentChar c = isBoaAlphaNum c || c == '_'

isBoaAlpha :: Char -> Bool
isBoaAlpha c =
  let c' = ord c in
      -- ord c [65,...,90] = [A-Z], ord c [97,..,122] = [a,..,z]
      (c' >= 65 && c' <= 90) || (c' >= 97 && c' <= 122)

isBoaAlphaNum :: Char -> Bool
isBoaAlphaNum c = isDigit c || isBoaAlpha c

isStrChar :: Char -> Bool
isStrChar c = isAscii c && isPrint c

-- Constants

reservedIdents :: [String]
reservedIdents = ["None", "True", "False", "for", "if", "in", "not"]
```

7.4 Tests - Test.hs

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

import BoaAST
import BoaParser

import Test.Tasty
import Test.Tasty.HUnit

main :: IO ()
main = defaultMain $ localOption (mkTimeout 1000000) tests

tests :: TestTree
tests = testGroup "Tests" [stmtsTests, numTests, strTest, identTest, noneTrueFalseTest, exprOperExprTest, notTest, parenthesisTest, identPExprzPTest, sqBrackTest, forClauseTest, commentsTest ,minimalTests]

assertFailure' a = case a of
  Left e -> return ()
  Right p -> assertFailure $ "Unexpected parse: " ++ show p

stmtsTests = testGroup "stmts tests"
  [ testCase "1" $ parseString "1" @?= Right [SExp (Const (IntVal 1))]
  , testCase "x=1" $ parseString "x=1" @?= Right [SDef "x" (Const (IntVal 1))]
  , testCase "1;1" $ parseString "1;1" @?= Right [SExp (Const (IntVal 1)), SExp (Const (IntVal 1))]
  , testCase "1;x=1" $ parseString "1;x=1" @?= Right [SExp (Const (IntVal 1)),SDef "x" (Const (IntVal 1))]
  , testCase "* 1;x=1;" $ assertFailure' $ parseString "1;x=1;"
  ]

numTests = testGroup "pNum tests"
  [ testCase "1" $ parseString "1" @?= Right [SExp (Const (IntVal 1))]
  , testCase "0" $ parseString "0" @?= Right [SExp (Const (IntVal 0))]
  , testCase "-0" $ parseString "-0" @?= Right [SExp (Const (IntVal 0))]
  , testCase "-1" $ parseString "-1" @?= Right [SExp (Const (IntVal (-1)))]
  , testCase "* -01" $ case parseString "-01" of Left e -> return (); Right p -> assertFailure $ "Unexpected parse: " ++ show p
  , testCase "* 01" $ assertFailure' $ parseString "01"
  , testCase "* 007" $  assertFailure' $ parseString "007"
  , testCase "* +7" $ assertFailure' $ parseString "+7"
  , testCase "* --7" $ assertFailure' $ parseString "--7"
  , testCase "   -1    " $ parseString "   -1" @?= Right [SExp (Const (IntVal (-1)))]
  ]

strTest = testGroup "pStr tests"
  [ testCase "'a\\\n b\\n\\\nc\\\n\\nd'" $ parseString "'a\\\n b\\n\\\nc\\\n\\nd'" @?= Right [SExp (Const (StringVal "a b\nc\nd"))]
  , testCase "* '\\t'" $ assertFailure' $ parseString "'\\t'"
  , testCase "* '\n'" $ assertFailure' $ parseString "'\n'"
```

```haskell
  , testCase "* '\t'" $ assertFailure' $ parseString "'\t'"
  , testCase "''" $ parseString "''" @?= Right [SExp (Const (StringVal ""))]
  , testCase "\t\n'Hello World'\t\n" $ parseString "\t\n'Hello World'\t\n" @?= Right
    [SExp (Const (StringVal "Hello World"))]
  , testCase "'\\\\'" $ parseString "'\\\\'" @?= Right [SExp (Const (StringVal "\\"))]
  , testCase "* 'a\nb'" $ assertFailure' $ parseString "'a\nb'"
  , testCase "'a\\nb'" $ parseString "'a\\nb'" @?= Right [SExp (Const (StringVal
    "ab"))]
  , testCase "x='Hello World!'" $ parseString "x='Hello World!'" @?= Right [SDef "x"
    (Const (StringVal "Hello World!"))]
  ]

identTest = testGroup "pIdent tests"
  [ testCase "x=2" $ parseString "x=2" @?= Right [SDef "x" (Const (IntVal 2))]
  , testCase "x=10//5" $ parseString "x=10//5" @?= Right [SDef "x" (Oper Div (Const
    (IntVal 10)) (Const (IntVal 5)))]
  , testCase "OneTo5=2" $  parseString "OneTo5=2" @?= Right [SDef "OneTo5" (Const (IntVal
    2))]
  , testCase "_One_To5_=2" $ parseString "_One_To5_=2" @?= Right [SDef "_One_To5_" (Const
    (IntVal 2))]
  , testCase "  \t\n_One_To5_  \t\n=  \t\n2  \t\n" $ parseString "  \t\n_One_To5_  \t\n=
    \t\n2  \t\n" @?= Right [SDef "_One_To5_" (Const (IntVal 2))]
  , testCase "* 1to5=2" $ assertFailure' $ parseString "1to5=2"
  , testCase "* None=2" $ assertFailure' $  parseString "None=2"
  , testCase "* True=2" $ assertFailure' $  parseString "True=2"
  , testCase "* False=2" $ assertFailure' $  parseString "False=2"
  , testCase "* for=2" $ assertFailure' $  parseString "for=2"
  , testCase "* if=2" $ assertFailure' $  parseString "if=2"
  , testCase "* in=2" $ assertFailure' $  parseString "in=2"
  , testCase "* not=2" $ assertFailure' $  parseString "not=2"
  ]

noneTrueFalseTest = testGroup "pNoneTrueFalseTest"
  [ testCase "x=None" $ parseString "x=None" @?= Right [SDef "x" (Const NoneVal)]
  , testCase "x=True" $ parseString "x=True" @?= Right [SDef "x" (Const TrueVal)]
  , testCase "x=False" $ parseString "x=False" @?= Right [SDef "x" (Const FalseVal)]
  , testCase "  \t\nx  \t\n=  \t\nFalse  \t\n" $ parseString "  \t\nx  \t\n=  \t\nFalse
    \t\n" @?= Right [SDef "x" (Const FalseVal)]
  ]

exprOperExprTest = testGroup "Expr Oper Expr"
  -- Test operators first
  [ testCase "1+1" $ parseString "1+1" @?= Right [SExp (Oper Plus (Const (IntVal 1))
    (Const (IntVal 1)))]
  , testCase "1-1" $ parseString "1-1" @?= Right [SExp (Oper Minus (Const (IntVal 1))
    (Const (IntVal 1)))]
  , testCase "1*1" $ parseString "1*1" @?= Right [SExp (Oper Times (Const (IntVal 1))
    (Const (IntVal 1)))]
  , testCase "1//1" $ parseString "1//1" @?= Right [SExp (Oper Div (Const (IntVal 1))
    (Const (IntVal 1)))]
```

```haskell
  , testCase "1%1" $ parseString "1%1" @?= Right [SExp (Oper Mod (Const (IntVal 1)) (Const (IntVal 1)))]
  , testCase "1==1" $ parseString "1==1" @?= Right [SExp (Oper Eq (Const (IntVal 1)) (Const (IntVal 1)))]
  , testCase "1!=1" $ parseString "1!=1" @?= Right [SExp (Not (Oper Eq (Const (IntVal 1)) (Const (IntVal 1))))]
  , testCase "1<1" $ parseString "1<1" @?= Right [SExp (Oper Less (Const (IntVal 1)) (Const (IntVal 1)))]
  , testCase "1<=1" $ parseString "1<=1" @?= Right [SExp (Not (Oper Greater (Const (IntVal 1)) (Const (IntVal 1))))]
  , testCase "1>1" $ parseString "1>1" @?= Right [SExp (Oper Greater (Const (IntVal 1)) (Const (IntVal 1)))]
  , testCase "1>=1" $ parseString "1>=1" @?= Right [SExp (Not (Oper Less (Const (IntVal 1)) (Const (IntVal 1))))]
  , testCase "1 in 1" $ parseString "1 in 1" @?= Right [SExp (Oper In (Const (IntVal 1)) (Const (IntVal 1)))]
  , testCase "1 not in 1" $ parseString "1 not in 1" @?= Right [SExp (Not (Oper In (Const (IntVal 1)) (Const (IntVal 1))))]
  -- Test Left Associativity ((1+2)+3)
  , testCase "1+2+3" $ parseString "1+2+3" @?= Right [SExp (Oper Plus (Oper Plus (Const (IntVal 1)) (Const (IntVal 2))) (Const (IntVal 3)))]
  -- Test precedende level of operators
  , testCase "1+2*3" $ parseString "1+2*3" @?= Right [SExp (Oper Plus (Const (IntVal 1)) (Oper Times (Const (IntVal 2)) (Const (IntVal 3))))]
  -- Test precedende level of operators and associativity (1+((2*3)/4))
  , testCase "1+2*3//4" $ parseString "1+2*3//4" @?= Right [SExp (Oper Plus (Const (IntVal 1)) (Oper Div (Oper Times (Const (IntVal 2)) (Const (IntVal 3))) (Const (IntVal 4))))]
  -- Test lt/gt
  , testCase  "2+2<1+1" $ parseString "2+2<1+1" @?= Right [SExp (Oper Less (Oper Plus (Const (IntVal 2)) (Const (IntVal 2))) (Oper Plus (Const (IntVal 1)) (Const (IntVal 1))))]
  , testCase "1+not 1" $ assertFailure' $ parseString "1+not 1"
  ]

notTest = testGroup "not Expr"
  [ testCase "not True" $ parseString "not True" @?= Right [SExp (Not (Const TrueVal))]
  , testCase "not 1" $ parseString "not 1" @?= Right [SExp (Not (Const (IntVal 1)))]
  , testCase "not not 1" $ parseString "not not 1" @?= Right [SExp (Not (Not (Const (IntVal 1))))]
  ]

parenthesisTest = testGroup "parenthesis Test"
  [ testCase "(1)" $ parseString "(1)" @?= Right [SExp (Const (IntVal 1))]
  , testCase "(  \n  \t 1  \n \t  )" $ parseString "(  \n  \t 1  \n \t  )" @?= Right [SExp (Const (IntVal 1))]
  , testCase "('Hello World')" $ parseString "('Hello World')" @?= Right [SExp (Const (StringVal "Hello World"))]
  , testCase "(_1hello_world_)" $ parseString "(_1hello_world_)" @?= Right [SExp (Var "_1hello_world_")]
```

```haskell
  , testCase "(None)" $ parseString "(None)" @?= Right [SExp (Const NoneVal)]
  , testCase "(True)" $ parseString "(True)" @?= Right [SExp (Const TrueVal)]
  , testCase "(False)" $ parseString "(False)" @?= Right [SExp (Const FalseVal)]
  , testCase "(not False)" $ parseString "(not False)" @?= Right [SExp (Not (Const FalseVal))]
  , testCase "((1))" $ parseString "((1))" @?= Right [SExp (Const (IntVal 1))]
  , testCase "(1+1)" $ parseString "(1+1)" @?= Right [SExp (Oper Plus (Const (IntVal 1)) (Const (IntVal 1)))]
  , testCase "(1<1)" $ parseString "(1<1)" @?= Right [SExp (Oper Less (Const (IntVal 1)) (Const (IntVal 1)))]
  , testCase "(((1)))" $ parseString "(((1)))" @?= Right [SExp (Const (IntVal 1))]
  , testCase "(range(10,2,3))" $ parseString "(range(10,2,3))" @?= Right [SExp (Call "range" [Const (IntVal 10),Const (IntVal 2),Const (IntVal 3)])]
  , testCase "([1,x,[1,2],'Hello World!'])" $ parseString "([1,x,[1,2],'Hello World!'])" @?= Right [SExp (List [Const (IntVal 1),Var "x",List [Const (IntVal 1),Const (IntVal 2)],Const (StringVal "Hello World!")])]
  , testCase "([x for x in z if b < y])" $ parseString "([x for x in z if b < y])" @?= Right [SExp (Compr (Var "x") [CCFor "x" (Var "z"),CCIf (Oper Less (Var "b") (Var "y"))])]
  ]

identPExprzPTest = testGroup "ident Paren Exprz Paren Test"
  [ testCase "range(1,2,3)" $ parseString "range(1,2,3)" @?= Right [SExp (Call "range" [Const (IntVal 1),Const (IntVal 2),Const (IntVal 3)])]
  , testCase "print(1,2,3)" $ parseString "print(1,2,3)" @?= Right [SExp (Call "print" [Const (IntVal 1),Const (IntVal 2),Const (IntVal 3)])]
  , testCase "print()" $ parseString "print()" @?= Right [SExp (Call "print" [])]
  , testCase "print \n \t (1,2,3)" $ parseString "print \n \t (1,2,3)" @?= Right [SExp (Call "print" [Const (IntVal 1),Const (IntVal 2),Const (IntVal 3)])]
  , testCase "True(1,2,3)" $ assertFailure' $ parseString "True(1,2,3)" -- reserved keyword
  ]

sqBrackTest = testGroup "Square Bracket Tests"
  [ testCase "[1,2,3]" $ parseString "[1,2,3]" @?= Right [SExp (List [Const (IntVal 1),Const (IntVal 2),Const (IntVal 3)])]
  , testCase "[1,2,'3']" $ parseString "[1,2,'3']" @?= Right [SExp (List [Const (IntVal 1),Const (IntVal 2),Const (StringVal "3")])]
  , testCase "[1]" $ parseString "[1]" @?= Right [SExp (List [Const (IntVal 1)])]
  , testCase "[]" $ parseString "[]" @?= Right [SExp (List [])]
  , testCase "[1,True,False,None,'Hello',[], [1,2]]" $ parseString "[1,True,False,None,'Hello',[], [1,2]]" @?= Right [SExp (List [Const (IntVal 1),Const TrueVal,Const FalseVal,Const NoneVal,Const (StringVal "Hello"),List [],List [Const (IntVal 1),Const (IntVal 2)]])]
  ]

forClauseTest = testGroup "For Clause Test"
  [ testCase "[x for y in z if u > 2]" $ parseString "[x for y in z if u > 2]" @?= Right [SExp (Compr (Var "x") [CCFor "y" (Var "z"),CCIf (Oper Greater (Var "u") (Const (IntVal 2)))])]
```

```haskell
  , testCase "[x\nfor\ty\tin\tz\tif\tu\t>\t2]" $ parseString "[x\nfor\ty\tin\tz\tif\tu\t>\t2]" @?= Right [SExp (Compr (Var "x") [CCFor "y" (Var "z"),CCIf (Oper Greater (Var "u") (Const (IntVal 2)))])]
  , testCase "[xfor y in z if u > 2]" $ assertFailure' $ parseString "[xfor y in z if u > 2]"
  , testCase "[x fory in z if u > 2]" $ assertFailure' $ parseString "[x fory in z if u > 2]"
  , testCase "[x for yin z if u > 2]" $ assertFailure' $ parseString "[x for yin z if u > 2]"
  , testCase "[x for y inz if u > 2]" $ assertFailure' $ parseString "[x for y inz if u > 2]"
  , testCase "[x for y in zif u > 2]" $ assertFailure' $ parseString "[x for y in zif u > 2]"
  , testCase "[x for y in z ifu > 2]" $ assertFailure' $ parseString "[x for y in z ifu > 2]"
  , testCase "[(x)for(y)in(z)if(u)]" $ assertFailure' $ parseString "[(x)for(y)in(z)if(u)]"
  ]

commentsTest = testGroup "Comments test"
  [ testCase "#Hello\tWorld!\nx" $ parseString "#Hello\tWorld!\nx" @?= Right [SExp (Var "x")]
  , testCase "#\nx" $ parseString "#\nx" @?= Right [SExp (Var "x")]
  , testCase "x #  \n" $ parseString "x #  \n" @?= Right [SExp (Var "x")]
  , testCase "#  \n#  \n#  \nx#  \n#  \n#  \n" $ parseString "#  \n#  \n#  \nx#  \n#  \n#  \n" @?= Right [SExp (Var "x")]
  , testCase "#  \n#  \n#  \nx#  \n#  \n#hej  " $ parseString "#  \n#  \n#  \nx#  \n#  \n#hej" @?= Right [SExp (Var "x")]
  ]

minimalTests = testGroup "Minimal tests" [
  testCase "simple success" $
    parseString "2 + two" @?=
      Right [SExp (Oper Plus (Const (IntVal 2)) (Var "two"))],

  testCase "Big program" $
    parseString "x = [y*2 for y in range(1,10,2) if y > 5] # Some comment\n" @?=
      Right [SDef "x" (Compr (Oper Times (Var "y") (Const (IntVal 2))) [CCFor "y" (Call "range" [Const (IntVal 1),Const (IntVal 10),Const (IntVal 2)]),CCIf (Oper Greater (Var "y") (Const (IntVal 5)))])],
  testCase "simple failure" $
    -- avoid "expecting" very specific parse-error messages
    case parseString "wow!" of
      Left e -> return ()  -- any message is OK
      Right p -> assertFailure $ "Unexpected parse: " ++ show p]
```