# Advanced programming – Assignment 3

**Design and implementation**

We decided to go with the "ReadP" library, as we have already used it in the warm-up and are more familiar with it than the "Parsec" library. Based on the warmup, we modified the grammar to solve problems such as disambiguation, associativity, precedence, factorization problems and left recursion. Since we worked with the "ReadP" library, we had to get rid of the recursion on the left. Also, to solve the problem of associativity and precedence, we had to solve the problem of left recursion. The implementation of the parser for the new grammar is quite simple, we use <|> and do sequences to distinguish between several possible parsers.

The "skip" function is the method that is most used by the other functions and is based on the functionality of the "skipSpaces" function from "ReadP", but has as an additional property the fact that it also ignores comments. Since the functionality of the skip method does not include the cases where there is a whitespace between the keywords and the case where the nonterminal is an expression and starts without whitespace and brackets, we added the "skipW" and "skipWB" functions.

The “pIdent” function uses the look method to determine if one of the identifiers is a keyword or not. It also checks if the first character of the string is also a letter or an underscore, the other characters can be alphanumeric or the underscore. So, the “pIdent” function uses the munch function.

**Assessment of the code**

We tried to assess as much as possible, and to test all possible input data. In total in the test suite there are 102 tests with different input data. Within the test suite we created multiple test groups to have some structure, we tried to group most test by category, but we also have a group called “Tests” where are cases that we didn’t yet assign to a category. In total there are 10 categories of groups:

1. Minimal tests
2. Tests
3. Test expresions
4. Test operator
5. Test operator associativity/precedence
6. Tests comments and whitespaces
7. Tests of ident
8. Tests of numConst
9. Tests of stringConst
10. Test parenthesis

Each test has a name that explain it, more or less, ex: “deep brackets []”, “Special character string”.

To run the test suite, you need to run in terminal “stack test”.

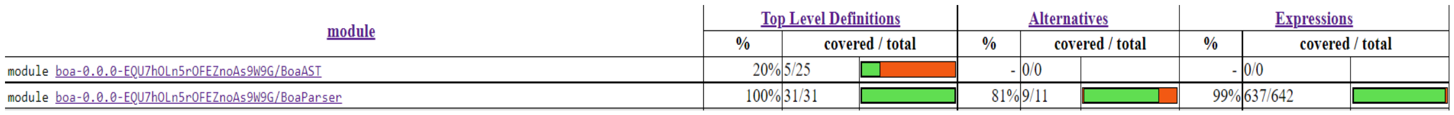
## Completeness

The code is 100% complete. Function “parseString” can parse any correct string, and in case that the syntax is not respected it throws an error message. All the syntax of Boa is implemented, all boa syntax has been implemented, so the input string can be any valid expression for boa.

## Correctness

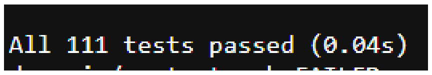
There aren’t any known bugs. The code works fine with all the test cases we created and with the online TA ones.

The correctness of the code was mainly tested with the test suite we created. In this test suite are 102 tests, and we tried to assess most of the cases. The coverage of the test is pretty good.



| module | Top Level Definitions | | | Alternatives | | | Expressions | | |
|---|---|---|---|---|---|---|---|---|---|
| | % | covered / total | | % | covered / total | | % | covered / total | |
| module boa-0.0.0-EQU7hOLn5rOFEZnoAs9W9G/BoaAST | 20% | 5/25 | 🟩🟥 | - | 0/0 | | - | 0/0 | |
| module boa-0.0.0-EQU7hOLn5rOFEZnoAs9W9G/BoaParser | 100% | 31/31 | 🟩 | 81% | 9/11 | 🟩🟥 | 99% | 637/642 | 🟩 |

We also did a sanity check with the online tool (https://find.incorrectness.dk/), and out of 111 tests, we passed them all.



```
All 111 tests passed (0.04s)
```

## Efficiency

The overall space complexity of the code is pretty good, we don’t know exactly the value, but in the case of deep brackets, it doesn’t take long to assess the string. About time complexity we are not sure how to measure it, but from what we saw from the tests and from the online tool they run fairly fast, so we think the time complexity should be fine. As for improving the code, we are not sure, what it can be done, as we tried to write the code in the most efficient way.

## Robustness

The code is robust, in situations where the input data may be correct from the Haskell point of view, but are not good as data for functions, then an error is returned. There are many possible cases when syntax/lexical can be right, but in this case, they are wrong, and these cases are treated and an error is thrown. Most common error message is “cannot parse” and it appears when there is an illegal input data. Besides that, there is also an error message “error”.

## Maintainability

The code is in good shape, all the repetitive code was transformed into functions, so instead of having a lot of copy pasted code, there are functions, that are easy to maintain and modify.  As for the test suits, most of the cases are copy pasted segments with changes.

From the point of view of comments, the code contains some.  The layout of the code is right all, the indentation is respected.

## Other

### 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 ::= TE'| "-"TE'".
--   E'::= "+" TE'| "-"TE'|^.
--   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)


-- E ::= TE'| "-"TE'".
pExp :: Parser Exp
pExp = do e <- pTerm; pExp' e
       <|> do symbol "-";  e <- pTerm ; pExp' (Negate e)

-- E'::= "+" TE'| "-"TE'|^.
pExp' :: Exp -> Parser Exp -- argument is "accumulator"
pExp' e1 = do ao <- pAddOp; e2 <- pTerm; pExp' (ao e1 e2)
           <|> return e1

--   T ::= num | "(" E ")" .
pTerm :: Parser Exp
pTerm = do n <- pNum; return $ Num n
        <|> do symbol "("; e <- pExp; symbol ")"; return e

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

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

pNum :: Parser Int
pNum = lexeme $ do
  ds <- many1 (satisfy isDigit)
  return $ read ds


pAddOp :: Parser (Exp -> Exp -> Exp)
pAddOp = do symbol "+"; return Add
         <|> do symbol "-"; return (\x y-> Add x  (Negate y))

parseString :: String -> Either ParseError Exp
parseString s = if null (readP_to_S pExp s)
  then
    Left "Parse Failed."
    else do
      let tmp = [x | x <- readP_to_S pExp s, snd x == ""]
      ;if null tmp
        then
          Left  "Parse Failed."
          else
            Right (fst (head tmp))
```

**BoaParser.hs**

```
-- Skeleton file for Boa Parser.
module BoaParser (ParseError, parseString) where

import BoaAST
import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))
import Data.Char
import Control.Monad
-- add any other other imports you need

type Parser a = ReadP a
type ParseError = String -- you may replace this

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

--works like skipSpaces but also skips comments
skip :: Parser ()
skip = do
  skipSpaces;
  s <- look
  when (not (null s) && head s == '#') $ do
    manyTill get (satisfy (== '\n') <|> do eof; return 'a')
    skip

--like skip, but there should be as a minimum one whitespace
skipW :: Parser ()
skipW = do
  s <- look
  if not (null s) && head s == '#'
    then skip
    else do
      munch1 isSpace
      skip

--Like skip, however there should be at the least one whitespace or the following
expression starts with a bracket
skipWB :: Parser ()
skipWB = do
  s <- look
  if not (null s) && (head s == '(' || head s == '[')
    then return ()
    else if head s == '#'
      then skip
      else do
        munch1 isSpace
        skip

--parses an identifier
--does not skip at the end, in order to make the use of skipW possible where needed
pIdent :: Parser String
pIdent = do
  s <- look
```

```haskell
  if any (\str -> take (length str) s == str && (null (drop (length str) s) ||
not(stringChar (s !! max 0 (length str))))) reserveWords
    then pfail
    else do
      c <- satisfy (\c -> isAlpha c || c=='_')
      cs <- munch stringChar
      return (c:cs)
  where
    stringChar :: Char -> Bool
    stringChar c = isAlphaNum c || c=='_'

--parses a numeric constant
pNumConst :: Parser Exp
pNumConst = do char '-'; pNumConst' "-"
  <|> pNumConst' ""

digits :: [Char]
digits = ['1','2','3','4','5','6','7','8','9']

pNumConst' :: String -> Parser Exp
pNumConst' b = do char '0'; return (Const (IntVal 0))
  <|> do
    c <- satisfy (`elem` digits)
    cs <- munch isNumber
    skip;
    return (Const (IntVal (read (b ++ c:cs))))

--parses a string constant
pStringConst :: Parser Exp
pStringConst = do
  char '\''
  str <- pStr ""
  skip;
  return (Const (StringVal str))

pStr :: String -> Parser String
pStr s = do char '\\'; char 'n'; pStr $ s ++ "\n"
  <|> do char '\\'; char '\''; pStr $ s ++ "'"
  <|> do char '\\'; char '\\'; pStr $ s ++ "\\"
  <|> do char '\\'; char '\n'; pStr s
  <|> do c <- satisfy (\c -> c /= '\\' && c /= '\'' && isPrint c); pStr $ s ++ [c]
  <|> do char '\''; return s

pStmts :: Parser [Stmt]
pStmts = do s <- pStmt; ss <- pStmt'; skip; return $ s:ss

pStmt' :: Parser [Stmt]
pStmt' = do char ';'; skip; pStmts
  <|> return []


pStmt :: Parser Stmt
pStmt = do i <- pIdent; skip; char '='; skip; e<-pExp; return $ SDef i e
  <|> do e <- pExp; return $ SExp e
```

```haskell
pExp :: Parser Exp
pExp = do string "not"; skipWB; e <- pExp; return (Not e)
  <|> pExpOrd


pExpOrd :: Parser Exp
pExpOrd = do
  e <- pExpAdd
  pExpOrd' e


pExpOrd' :: Exp -> Parser Exp
pExpOrd' e = do f <- helper "=="; return (Oper Eq e f)
  <|> do f <- helper "!="; return (Not (Oper Eq e f))
  <|> do f <- helper "<"; return (Oper Less e f)
  <|> do f <- helper "<="; return (Not (Oper Greater e f))
  <|> do f <- helper ">"; return (Oper Greater e f)
  <|> do f <- helper ">="; return (Not (Oper Less e f))
  <|> do f <- helperOne "in"; return (Oper In e f)
  <|> do string "not"; skipW; f <- helperOne "in"; return (Not (Oper In e f))
  <|> return e
  where
    helper s = do string s; skip; pExpAdd
    helperOne s = do string s; skipWB; pExpAdd

pExpAdd :: Parser Exp
pExpAdd = do e <- pExpMul; pExpAdd' e


pExpAdd' :: Exp -> Parser Exp
pExpAdd' e = do char '+'; skip; f <- pExpMul; pExpAdd' (Oper Plus e f)
  <|> do char '-'; skip; f <- pExpMul; pExpAdd' (Oper Minus e f)
  <|> return e


pExpMul :: Parser Exp
pExpMul = do e <- pExpTerm; pExpMul' e

pExpMul' :: Exp -> Parser Exp
pExpMul' e = do char '*'; skip; f <- pExpTerm; pExpMul' (Oper Times e f)
  <|> do char '%'; skip; f <- pExpTerm; pExpMul' (Oper Mod e f)
  <|> do string "//"; skip; f <- pExpTerm; pExpMul' (Oper Div e f)
  <|> return e


pExpTerm :: Parser Exp
pExpTerm = pNumConst
  <|> pStringConst
  <|> do string "None"; skip; return (Const NoneVal)
  <|> do string "True"; skip; return (Const TrueVal)
  <|> do string "False"; skip; return (Const FalseVal)
  <|> do i <- pIdent; skip; pExpI i
  <|> do char '('; skip; e <- pExp; char ')'; skip; return e
  <|> do char '['; skip; e <- pExpB; char ']'; skip; return e
```

```haskell
pExpI :: String -> Parser Exp
pExpI s = do char '('; skip; e <- pExpz s; char ')'; skip; return e
  <|> return (Var s)

pExpz :: String -> Parser Exp
pExpz s = do e <- pExp; pExpzs s [e]
  <|> return (Call s [])

pExpzs :: String -> [Exp] -> Parser Exp
pExpzs s es = do char ','; skip; e <- pExp; pExpzs s (es++[e])
  <|> return (Call s es)

pExpB :: Parser Exp
pExpB = do e <- pExp; pExpB' e
  <|> return (List [])

pExpB' :: Exp -> Parser Exp
pExpB' e = pExps [e]
  <|> do f <- pForClause; pClausez e [f]

pExps :: [Exp] -> Parser Exp
pExps es = do char ','; skip; e <- pExp; pExps (es ++ [e])
  <|> return (List es)

pForClause :: Parser CClause
pForClause = do
  string "for"
  skipW
  i <- pIdent
  skipWB
  string "in"
  skipWB
  e <- pExp
  skip;
  return (CCFor i e)

pIfClause :: Parser CClause
pIfClause = do
  string "if"
  skipWB
  e <- pExp
  skip;
  return (CCIf e)

pClausez :: Exp -> [CClause] -> Parser Exp
pClausez e cs = do f <- pForClause; pClausez e (cs++[f])
  <|> do i <- pIfClause; pClausez e (cs++[i])
  <|> return (Compr e cs)

parseString :: String -> Either ParseError Program
parseString s = case readP_to_S (do skip; a <- pStmts; eof; return a) s of
  [] -> Left "cannot parse"
  [(a,_)] -> Right a
  _ -> error "Error"
```

**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 = testGroup "Tests"
  [
  testGroup "Minimal tests" [
    testCase "simple success" $    parseString "2 + two" @?= Right [SExp (Oper Plus (Const (IntVal 2))
(Var "two"))],
    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
  ],
  testGroup "Tests" [
    testCase "Int value" $ parseString "1" @?= Right [SExp (Const (IntVal 1))],
    testCase "Big Int value" $ parseString "122222222222222" @?= Right [SExp (Const (IntVal
122222222222222))],
    testCase "String value" $ parseString "a" @?= Right [SExp (Var "a")],
    testCase "Long String value" $ parseString "qwertyuiopasdfghjklzxcvbnm" @?= Right [SExp (Var
"qwertyuiopasdfghjklzxcvbnm")],
    testCase "String value plus Int value" $ parseString "qwertyuiopasdfghjklzxcvbnm123_231124+2" @?=
Right [SExp (Oper Plus (Var "qwertyuiopasdfghjklzxcvbnm123_231124") (Const (IntVal 2)))],
    testCase "String value with operation with comments in between" $ parseString
"ab+2//3#test\n#test\n%3" @?= Right [SExp (Oper Plus (Var "ab") (Oper Mod (Oper Div (Const (IntVal 2))
(Const (IntVal 3))) (Const (IntVal 3))))],
    testCase "Empty string" $ parseString " " @?= (Left "cannot parse"),
    testCase "Special character string" $ parseString "'!@'" @?= Right [SExp (Const (StringVal "!@"))],


    testCase "Num, Stmts ;" $ parseString "-1234; 1" @?= Right [SExp (Const (IntVal (-1234))),SExp (Const
(IntVal 1))],
    testCase "String" $ parseString "'\\'asf124\\\n\\n  f'" @?= Right [SExp (Const (StringVal
"'asf124\n  f"))],
    testCase "String failure" $
      case parseString "'\'123" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
    testCase "Exp indent ()" $ parseString "print (123)" @?= Right [SExp (Call "print" [Const (IntVal
123)])],
    testCase "CClause" $ parseString "[a for k in 1 if c]" @?= Right [SExp (Compr (Var "a") [CCFor "k"
(Const (IntVal 1)),CCIf (Var "c")])],
    testCase "Stmt indent" $ parseString "a = 1" @?= Right [SDef "a" (Const (IntVal 1))],
    testCase "Call 0 args" $ parseString "f()" @?= Right [SExp (Call "f" [])],
    testCase "Call 1 args" $ parseString "g(True)" @?= Right [SExp (Call "g" [Const TrueVal])],
    testCase "Call 2 args" $ parseString "h(True,k())" @?= Right [SExp (Call "h" [Const TrueVal,Call "k"
[]])],
    testCase "Strange ranges" $ parseString "[range(), range(1,2,3,4), range(None)]" @?= Right [SExp (List
[Call "range" [],Call "range" [Const (IntVal 1),Const (IntVal 2),Const (IntVal 3),Const (IntVal 4)],Call
"range" [Const NoneVal]])],
    testCase "Def" $ parseString "x=y " @?= Right [SDef "x" (Var "y")],
    testCase "Seq" $ parseString "x;y=z;u;v=x " @?= Right [SExp (Var "x"),SDef "y" (Var "z"),SExp (Var
"u"),SDef "v" (Var "x")],
    testCase "Eq" $ parseString "x=y==z" @?= Right [SDef "x" (Oper Eq (Var "y") (Var "z"))],
    testCase "Seq1" $ parseString "x;x in [1,2,3]" @?= Right [SExp (Var "x"),SExp (Oper In (Var "x") (List
[Const (IntVal 1),Const (IntVal 2),Const (IntVal 3)]))],
    testCase "Seq2 " $ parseString "[1,2,3];[4,5,6];[7,8,9]" @?= Right [SExp (List [Const (IntVal 1),Const
(IntVal 2),Const (IntVal 3)]),SExp (List [Const (IntVal 4),Const (IntVal 5),Const (IntVal 6)]),SExp (List
[Const (IntVal 7),Const (IntVal 8),Const (IntVal 9)])]
    ],
  testGroup "Test expresions" [
    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 "Exp Not" $ parseString "not a" @?= Right [SExp (Not (Var "a"))],
    testCase "Exp no space Not -> Var" $ parseString "nota" @?= Right [SExp (Var "nota")],
    testCase "Comprehension for in in" $ parseString "[x for y in z in u]" @?= Right [SExp (Compr (Var
"x") [CCFor "y" (Oper In (Var "z") (Var "u"))])],
    testCase "Comprehension " $ parseString "[x for y in z if u for a in []]" @?= Right [SExp (Compr (Var
"x") [CCFor "y" (Var "z"),CCIf (Var "u"),CCFor "a" (List [])])],
    testCase "comprehension for ynot in " $ parseString "[x for ynot in z]" @?= Right [SExp (Compr (Var
"x") [CCFor "ynot" (Var "z")])]

  ],
  testGroup "Test operator" [
    testCase "Oper +" $ parseString "1 + 2 " @?= Right [SExp (Oper Plus (Const (IntVal 1)) (Const (IntVal
2)))],
    testCase "Oper +" $ parseString "1 + -2 " @?= Right [SExp (Oper Plus (Const (IntVal 1)) (Const (IntVal
(-2))))],
    testCase "Oper -" $ parseString "1 - 2 " @?= Right [SExp (Oper Minus (Const (IntVal 1)) (Const (IntVal
2)))],
    testCase "Oper -" $ parseString "1 - -2 " @?= Right [SExp (Oper Minus (Const (IntVal 1)) (Const
(IntVal (-2))))],
    testCase "Oper *" $ parseString "1*2 " @?= Right [SExp (Oper Times (Const (IntVal 1)) (Const (IntVal
2)))],
    testCase "Oper *" $ parseString "1*-2 " @?= Right [SExp (Oper Times (Const (IntVal 1)) (Const (IntVal
(-2))))],
    testCase "Oper //" $ parseString "1//2 " @?= Right [SExp (Oper Div (Const (IntVal 1)) (Const (IntVal
2)))],
    testCase "Oper //" $ parseString "1//-2 " @?= Right [SExp (Oper Div (Const (IntVal 1)) (Const (IntVal
(-2))))],
    testCase "Oper %" $ parseString "1%2 " @?= Right [SExp (Oper Mod (Const (IntVal 1)) (Const (IntVal
2)))],
    testCase "Oper %" $ parseString "1%-2 " @?= Right [SExp (Oper Mod (Const (IntVal 1)) (Const (IntVal (-
2))))],
    testCase "Oper + - * // %" $ parseString "[x+y,x-y,x*y,x//y,x%y] " @?= Right [SExp (List [Oper Plus
(Var "x") (Var "y"),Oper Minus (Var "x") (Var "y"),Oper Times (Var "x") (Var "y"),Oper Div (Var "x") (Var
"y"),Oper Mod (Var "x") (Var "y")])],


    testCase "Oper + * %" $ parseString "1 + 2 * 3 % 4" @?= Right [SExp (Oper Plus (Const (IntVal 1))
(Oper Mod (Oper Times (Const (IntVal 2)) (Const (IntVal 3))) (Const (IntVal 4))))],
    testCase "Oper - // ()" $ parseString "1 - 2 // (1+1)" @?= Right [SExp (Oper Minus (Const (IntVal 1))
(Oper Div (Const (IntVal 2)) (Oper Plus (Const (IntVal 1)) (Const (IntVal 1)))))],
    testCase "Oper not in, []" $ parseString "a not in [1, 2]" @?= Right [SExp (Not (Oper In (Var "a")
(List [Const (IntVal 1),Const (IntVal 2)])))],


    testCase "Oper <" $ parseString "1<2 " @?= Right [SExp (Oper Less (Const (IntVal 1)) (Const (IntVal
2)))],
    testCase "Oper ==" $ parseString "1==2 " @?= Right [SExp (Oper Eq (Const (IntVal 1)) (Const (IntVal
2)))],
    testCase "Oper >" $ parseString "1==2 " @?= Right [SExp (Oper Eq (Const (IntVal 1)) (Const (IntVal
2)))],
    testCase "Oper >=" $ parseString "a >= 6" @?= Right [SExp (Not (Oper Less (Var "a") (Const (IntVal
6))))],
    testCase "Oper <=" $ parseString "1<=2 " @?= Right [SExp (Not (Oper Greater (Const (IntVal 1)) (Const
(IntVal 2))))],
    testCase "Oper < and >" $ parseString "1 < a > 3" @?= (Left "cannot parse"),
    testCase "Oper >= and ==" $
    case parseString "a >= 6 == 6" of
      Left e -> return ()
      Right p -> assertFailure $ "Unexpected parse: " ++ show p,
    testCase "Oper == < > in" $ parseString "[x==y,x<y,x>y,x in y] " @?= Right [SExp (List [Oper Eq (Var
"x") (Var "y"),Oper Less (Var "x") (Var "y"),Oper Greater (Var "x") (Var "y"),Oper In (Var "x") (Var
"y")])],
    testCase "Oper  != <= >= not in" $ parseString "[x!=y,x>=y,x<=y,x not in y] " @?= Right [SExp (List
[Not (Oper Eq (Var "x") (Var "y")),Not (Oper Less (Var "x") (Var "y")),Not (Oper Greater (Var "x") (Var
"y")),Not (Oper In (Var "x") (Var "y"))])]

  ],
    testGroup "Test operator associativity/precedence" [
      testCase "associativity of add" $ parseString "1+2+3" @?= (Right [SExp (Oper Plus (Oper Plus (Const
(IntVal 1)) (Const (IntVal 2))) (Const (IntVal 3)))]),
      testCase "associativity of mul" $ parseString "1*2*3" @?= (Right [SExp (Oper Times (Oper Times
(Const (IntVal 1)) (Const(IntVal 2))) (Const (IntVal 3)))]),
      testCase "associativity of div/mod" $ parseString "1//2%3" @?= (Right [SExp (Oper Mod (Oper Div
(Const (IntVal 1)) (Const(IntVal 2))) (Const (IntVal 3)))]),
      testCase "associativity of minus" $ parseString "1-2-3" @?= (Right [SExp (Oper Minus (Oper Minus
(Const (IntVal 1)) (Const(IntVal 2))) (Const (IntVal 3)))]),
      testCase "precedence test" $ parseString "not 1+2*3-(not 4//5+6)" @?= (Right[SExp (Not(Oper Minus
(Oper Plus (Const (IntVal 1)) (Oper Times (Const (IntVal 2)) (Const (IntVal 3)))) (Not (Oper Plus (Oper
Div (Const (IntVal 4)) (Const (IntVal 5))) (Const (IntVal 6))))))]),
      testCase "no associativity of <,>,==" $ parseString "1 < 2 > 3 == 4" @?= (Left "cannot parse"),
      testCase "no associativity of <=,>=,==" $ parseString "1 <= 2 >= 3 == 4" @?= (Left "cannot parse")
      ],
    testGroup "Tests comments and whitespaces" [
      testCase "missing whitespace between keywords" $ parseString "a notin b" @?= (Left "cannot parse"),
      testCase "no whitespace, but bracket" $ parseString "not(False)" @?= (Right [SExp (Not (Const
FalseVal))]),
      testCase "Comments" $ parseString "True#test\n#test1\t\n;False" @?= (Right [SExp (Const
TrueVal),SExp (Const FalseVal)]),
      testCase "Comments at eof" $ parseString "True#test" @?= (Right [SExp (Const TrueVal)]),
      testCase "empty comment" $ parseString "True;#\nFalse" @?= (Right [SExp (Const TrueVal),SExp (Const
FalseVal)]),
      testCase "skipping newlines" $ parseString "True\n\n;\nFalse" @?= (Right [SExp (Const TrueVal),SExp
(Const FalseVal)]),
      testCase "skipping tabs" $ parseString "tab\t\t\t;False" @?= (Right [SExp (Var "tab"),SExp (Const
FalseVal)]),
      testCase "comment as one whitespace" $ parseString "not#comment\ncool" @?= (Right [SExp (Not (Var
"cool"))]),
      testCase "Tabs and newlines" $ parseString "\t\n [\t \n] \n\t" @?= Right [SExp (List [])],
      testCase "Interspersed whitespaces" $ parseString " x = ( 2 ) ; print ( None == [ y , z ] , not u )
" @?= Right [SDef "x" (Const (IntVal 2)),SExp (Call "print" [Oper Eq (Const NoneVal) (List [Var "y",Var
"z"]),Not (Var "u")])],
      testCase "No whitespaces" $ parseString "[(x)not\tin(not(y)),[(x)for\ty\tin[z]if(u)]]" @?= Right
[SExp (List [Not (Oper In (Var "x") (Not (Var "y"))),Compr (Var "x") [CCFor "y" (List [Var "z"]),CCIf (Var
"u")]])],
      testCase "Comment as separator" $ parseString "not#foo\nx" @?= Right [SExp (Not (Var "x"))],
      testCase "Only comments" $ parseString "#com#com1#com2#com3" @?= (Left "cannot parse"),
      testCase "Large whitespaces" $ parseString
"                                                       x
        " @?= Right [SExp (Var "x")],
      testCase "Many comments" $ parseString
"#  \n#  \n#  \n#  \n#  \n#  \n#  \n#  \n#  \nx#  \n#  \n#  \n#  \n#  \n#  \n#  \n#  \n#  \n"
@?= Right [SExp (Var "x")]
      ],
    testGroup "Tests of ident" [
      testCase "keyword as identifier" $ parseString "in = out" @?= (Left "cannot parse"),
      testCase "keyword in identifier" $ parseString "inside = outside" @?= (Right [SDef "inside" (Var
"outside")]),
      testCase "starting with _" $ parseString "_underscore = UP" @?= (Right [SDef "_underscore" (Var
"UP")]),
      testCase "starting with number" $ parseString "112 = alarm" @?= (Left "cannot parse"),
      testCase "numbers, underscore and letters" $ parseString "_abc9 < Xyzw100_" @?= (Right [SExp (Oper
Less (Var "_abc9") (Var "Xyzw100_"))])
      ],
    testGroup "Tests of numConst" [
      testCase "minus zero '-0'" $ parseString "-0" @?= (Right [SExp (Const (IntVal 0))]),
      testCase "plus in front of number '+0'" $ parseString "+0" @?= (Left "cannot parse"),
      testCase "wrong number format '1.0'" $ parseString "1.0" @?= (Left "cannot parse"),
      testCase "space between minus and number '- 4'" $ parseString "- 4" @?= (Left "cannot parse"),
      testCase "starting with zeros '007'" $ parseString "007" @?= (Left "cannot parse")
      ],
    testGroup "Tests of stringConst" [
      testCase "'basic string'" $ parseString "'basic string'" @?= (Right [SExp (Const (StringVal "basic
string"))]),
      testCase "'a\\nb'" $ parseString "'a\\\nb'" @?= (Right [SExp (Const (StringVal "ab"))]),
      testCase "'\\\\'" $ parseString "'\\\\'" @?= (Right [SExp (Const (StringVal "\\"))]),
      testCase "'a#bc'" $ parseString "'a#bc'" @?= (Right [SExp (Const (StringVal "a#bc"))]),
      testCase "'\\''" $ parseString "'\\''" @?= (Right [SExp (Const (StringVal "'"))]),
      testCase "'\\x'" $ parseString "'\\x'" @?= (Left "cannot parse"),
      testCase "Simple escapes" $ parseString "'a\\'b\\\\c\\nd'" @?= Right [SExp (Const (StringVal
"a'b\\c\nd"))],
      testCase "Escaped newlines" $ parseString "'a\\\n b\\n\\\nc\\\n\\nd'" @?= Right [SExp (Const
(StringVal "a b\nc\nd"))]
      ],
    testGroup "Test parenthesis" [
      testCase "brackets []" $ parseString "[1]" @?= (Right [SExp (List [Const (IntVal 1)])]),
      testCase "brackets ()" $ parseString "(1)" @?= (Right [SExp (Const (IntVal 1))]),
      testCase "deep brackets []" $ parseString "[[[[[[[[[[[x]]]]]]]]]]]" @?= (Right [SExp (List [List
[List [List [List [List [List [List [List [List [Var "x"]]]]]]]]]]])]),
      testCase "very deep brackets []" $ parseString
"[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[[x]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]" @?= (Right [SExp (List [List [List
[List [List [List [List [List [List [List [List [List [List [List [List [List [List [List [List [List
[List [List [List [List [List [List [List [List [List [List [List [List [List [Var
"x"]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]]])]),
      testCase "deep brackets ()" $ parseString "(((((((((((x)))))))))))" @?= (Right [SExp (Var "x")]),
      testCase "very deep brackets ()" $ parseString
"(((((((((((((((((((((((((((((((((x)))))))))))))))))))))))))))))))))" @?= (Right [SExp (Var "x")])
      ]
    ]
```