Advanced Programming - Assignment 2

KuNET Id: vsf505

September 2020

1 Grammar

The given grammar had to be edited in order to solve disambiguation, precedence, associativity and left recursion/factoring issues. In the following section, i will try to explain the changes with one example for each.
In order to be able to work with the ReadP library, the grammar should not have any left-recursive productions. That is why a the production 'Expr ::= Expr Oper Expr' has to be modified. Fortunately, this issue could be solved in the same step as solving the precedence issue.
Solving the precedence issue of operators has been done the same way as it is shown in the lecture slides. The 'Expr' nonterminal and its productions are being split up into several new productions. Since not has the loosest binding, the basic 'Exp'-nonterminal in the new grammar can be derived into 'ExpOrd' which can be derived into 'ExpAdd' which can be derived into 'ExpMul' which can be derived into 'ExpTerm'.
Some of the stated nonterminals had to be "split up" (e.g. ExpAdd) into two nonterminals (e.g. ExpAdd, ExpAdd') in order to solve the associativity issue. As it was done in the lecture and also stated in the assignment text, mathematical expressions are left-associative. As stated in the lecture slides, solving the associativity also means solving a left-recursion and ends up in a transformation like this one:

```
ExpAdd ::= ExpMul OperAdd ExpAdd
=>  ExpAdd  ::= ExpMul ExpAdd'
    ExpAdd' ::= OpAdd ExpMul ExpAdd' | ϵ
```

The *Stmts*-nonterminal is a good example to explain the left factoring, i just used the procedure that has been used in the slides to minimize the disambiguation of the grammar. Since ReadP is a backtracking parser, this would not have been necessary for the code to work, but it reduces the work for the parser and makes the implementation of the grammar as code easier in my opinion.

```
Stmts ::= Stmt | Stmt ';' Stmts
=>  Stmts ::= Stmt Stmt'
    Stmt' ::= ';' Stmts | ϵ
```

The transformations lead to this final grammar:

Listing 1: Transformed grammar

```
Stmts      ::= Stmt Stmt'
Stmt'      ::= ";" Stmts | ϵ
Stmt       ::= ident "=" Exp
             | Exp
Exp        ::= "not" Exp
             | ExpOrd
ExpOrd     ::= ExpAdd ExpOrd'
ExpOrd'    ::= OpOrd ExpAdd | ϵ
OpOrd      ::= "==" | "!=" | "<" | "<=" | ">" | ">=" | "in" | "not" "in"
ExpAdd     ::= ExpMul ExpAdd'
ExpAdd'    ::= OpAdd ExpMul ExpAdd' | ϵ
OpAdd      ::= "+" | "-"
ExpMul     ::= ExpTerm ExpMul'
ExpMul'    ::= OpMul ExpTerm ExpMul' | ϵ
OpMul      ::= "*" | "//" | "%"
ExpTerm    ::= numConst
             | stringCst
             | "None"
             | "True"
             | "False"
             | ident ExpI
             | "(" Exp ")"
             | "[" ExpB "]"
ExpI       ::= "(" Expz ")" | ϵ
Expz       ::= Exp Expzs | ϵ
Expzs      ::= "," Exp Expzs | ϵ
ExpB       ::= Exp ExpB' | ϵ
ExpB'      ::= Exps
ExpI       ::= "(" Expz ")" | ϵ
Expz       ::= Exp Expzs | ϵ
Expzs      ::= "," Exp Expzs | ϵ
ExpB       ::= Exp ExpB' | ϵ
ExpB'      :: Exps | ForClause Clausez
Exps       ::= "," Exp Exps | ϵ
ForClause  ::= "for" ident "in" Exp
IfClause   ::= "if" Exp
Clausez    ::= ForClause Clausez | IfClause | ϵ
```

'ident', 'numConst', 'stringCst' are counted as terminals here.

2 Implementation

I decided to work with the ReadP library. I chose this library because i felt that backward compatibility could be necessary in some of the cases and because the alternative that is being used in the course, Parsec, makes writing many alternatives much more difficult since you had to use several 'try' statements.
The actual parser implementation of the grammar is pretty much straight forward using do statements and the $< | >$ operator to distinguish between multiple possible parsers.
The readString method is implemented using the same structure as shown in the lecture slides about ReadP.
As the grammar has been transformed to a parser-friendly grammar and the implementation of it is not that difficult, there are still some more things to do: The *skip* method is the function in the code that is being used by almost all other functions. It works like the skipSpaces function in the ReadP library but adds some functionality in order to support comments as specified in the assignment text. Between keywords, there has to be at least one whitespace but if the following nonterminal in the grammar is an expression, then no whitespace and a bracket starting the expression is allowed as well. That is why i implemented two more variants of skip, namely skipOne and skip1.
The only thing left now is the implementation and recognition of identifiers, numeric and string constants.
The pIdent function parses any identifier following the rules of the assignment. It first uses the look function of pRead in order to check whether the identifier equals one of the keywords and is therefore forbidden. The first letter of any identifier has to be a letter or a '_'. All other letters of an identifier can be alphanumeric or underscores. The function therefore uses the munch function. Numerical constants can be expressed with the following regular expression:

$$(\epsilon|-)(0|(1..9)(0..9)^*)$$

The parser implements the recognition of this regular expression pretty much straight forward.
Last but not least is the parsing of string constants. The implementation is again very intuitive. Every character is analyzed on its own and every time all allowed escape characters are being checked.

The OnlineTA mentions some possible redundancy in the code that could be fixed using an auxiliary function but in my opinion that makes the code much worse readable.
The parser uses the ReadP library with which it is rather difficult to produce good error messages. Therefore the parser only returns "cannot parse" if the given input does not have the correct syntax.

3 Testing

In order to test the correctness of my Boa parser, I used QuickCheck to generate arbitrary programs and did some unit tests to test the behavior in special cases. In the Test.hs file, you can find a (more or less meaning full) instantiation of Arbitrary for programs (including expressions, values, statements, operations...). Most of the programs that are being generated would end up in an error if you would interpret them but they are syntactically correct and therefore good enough to test the behavior of my parser. In order to test the parser, the arbitrary program is being printed and then used as input to parseString. The resulting program is then compared with the original one.
The tester also tests the (left) associativity of all operations and whether the precedendes are handled correctly.
The skip (and skip1/skipOne) function is being tested with some tests including comments.
Last but not least, the parsers for identifiers, numeric constants and string constants are being tested with some specific input including escape characters for string constants, leading zeros for numbers and keywords as part of identifiers. The exact test cases can be found in the test file itself (5.2).

4 Conclusion

My Boa parser passes all of my tests and all of the tests of the OnlineTA. It is able to ignore comments and throws exceptions if the syntax of the given program can not be parsed. The error messages are not very helpful and could be improved but writing better error messages using a backtracking parser is quite difficult and not required in this assignment. My testing that uses arbitrary instances of programs and specific unit tests should be sufficient to prove the correctness of my parser.

5 Code

5.1 BoaParser.hs

```haskell

module BoaParser (ParseError, parseString) where

import BoaAST
import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))
import Data.Char
import Control.Monad

type Parser a = ReadP a
```

```haskell
type ParseError = String

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 "oops, my grammar is ambiguos"

--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 must be at least one whitespace
skip1 :: Parser ()
skip1 = do
  s <- look
  if not (null s) && head s == '#'
    then skip
    else do
      munch1 isSpace
      skip

--like skip, but there must be at least one whitespace or the next expression starts with a bracket
skipOne :: Parser ()
skipOne = 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 skip1 possible where needed
pIdent :: Parser String
```

```haskell
pIdent = do
  s <- look
  if any (\str -> take (length str) s == str && (null (drop (length str) s) || not(stringChar (s !! max 0 (length str))))) ["None","True","False","for","if","in","not"]
    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' ""

pNumConst' :: String -> Parser Exp
pNumConst' b = do char '0'; return (Const (IntVal 0))
  <|> do
    c <- satisfy (\c -> c `elem` ['1','2','3','4','5','6','7','8','9'])
    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

```

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

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

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

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

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

--ExpOrd'      ::= OpOrd ExpAdd |
--OpOrd        ::= "==" | "!=" | "<" | "<=" | ">" | ">=" | "in" | "not in"
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"; skip1; f <- helperOne "in"; return (Not (Oper In e f))
  <|> return e
  where
    helper s = do string s; skip; pExpAdd
```

```
    helperOne s = do string s; skipOne; pExpAdd

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

--ExpAdd'    ::= OpAdd ExpMul ExpAdd' |
--OpAdd      ::= "+" | "-"
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

--ExpMul     ::= ExpTerm ExpMul'
--ExpMul'    ::= OpMul ExpTerm ExpMul' |
--OpMul      ::= "*" | "//" | "%"
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

--ExpTerm    ::= numConst
--              | stringCst
--              | "None"
--              | "True"
--              | "False"
--              | ident ExpI
--              | "(" Exp ")"
--              | "[" ExpB "]"
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) --skip1
  <|> do i <- pIdent; skip; pExpI i
```

```haskell
  <|> do char '('; skip; e <- pExp; char ')'; skip; return e
  <|> do char '['; skip; e <- pExpB; char ']'; skip; return e

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

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

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

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

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

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

--ForClause ::= "for" ident "in" Exp
pForClause :: Parser CClause
pForClause = do
  string "for"
  skip1
  i <- pIdent
  skipOne
  string "in"
```

```
  skipOne
  e <- pExp
  skip;
  return (CCFor i e)

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

--Clausez ::= ForClause Clausez | IfClause Clausez |
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)
```

5.2 Test.hs

```
import BoaAST
import BoaParser
import Test.Tasty
import Test.Tasty.HUnit
import Test.Tasty.QuickCheck
import Debug.Trace

-----------generating more or less meaning full arbitrary programs-----------------

asciiLetter = elements (['a'..'z']++['A'..'Z'])

--arbitrary operation
instance Arbitrary Op where
  arbitrary = oneof $ map return [Plus, Minus, Times, Div, Mod, Eq, Less, Greater, In]

--arbitrary value
instance Arbitrary Value where
  arbitrary = oneof [return NoneVal, return TrueVal, return FalseVal, randIntVal, randStringVal]
    where
      randIntVal = do
```

```
        i <- arbitrary
        return $ IntVal i
      randStringVal = do
        n <- choose (1,4)
        s <- vectorOf n asciiLetter
        return $ StringVal s
        --has been removed, since parser always [] as expression and not as value
      {-randListVal i = do
        n <- choose (1,4)
        xs <- vectorOf n $ arb $ i `div`2
        return $ ListVal xs-}

--arbitrary expression
arbitraryExp :: [String] -> Gen Exp
arbitraryExp xs = sized (\x -> arb xs x)
  where
    arb :: [VName] -> Int -> (Gen Exp)
    arb xs 0 = oneof [randConst, randVar xs]
    arb xs n = oneof [randConst, randVar xs, randOper xs n, randNot xs n, randList xs n, randCall xs n , randCompr xs n]
    randConst = do
      v <- arbitrary
      return $ Const v
    randVar xs = if length xs == 0
      then randConst
      else do
        v <- elements xs
        return $ Var v
    randOper xs n = do
      o <- arbitrary
      e <- arb xs (n`div`2)
      f <- arb xs (n`div`2)
      return $ Oper o e f
    randNot xs n = do
      e <- arb xs (n`div`2)
      return $ Not e
    randList xs n = do
      i <- choose (0,3)
      ys <- vectorOf i $ arb xs $ n `div` 2
      return $ List ys
    randCall xs n = oneof [randCallPrint xs n, randCallRange xs n, randCallDif]
    randCallDif = do
      s <- asciiLetter
```

```
      return $ Call [s] []
    randCallPrint xs n = do
      i <- choose (0,4)
      ys <- vectorOf i $ arb xs $ n `div` 2
      return $ Call "print" ys
    randCallRange _ _ = do
      i <- choose (0,4)
      ys <- vectorOf i arbitInt
      return $ Call "range" ys
        where
          arbitInt = do
            j <- arbitrary :: Gen Int
            return $ Const $ IntVal j
    randCompr xs n = do
      (CC x e) <- randFor xs $ n`div`2
      i <- choose (0,3)
      ys <- randCC (x:xs) n i
      f <- arb xs (n`div`2)
      return $ Compr f (e++ys)
        where
          randFor :: [String] -> Int -> Gen CC
          randFor xs n = do
            x <- asciiLetter
            e <- randList' xs n
            return $ CC [x] $ [CCFor [x] e]
          randIf :: [String] -> Int -> Gen CC
          randIf xs n = do
            e <- arb xs n
            return $ CC "" $ [CCIf e]
          randList' xs n = oneof [randList xs n, randCallRange xs n]
          randCC :: [String] -> Int -> Int -> Gen [CClause]
          randCC _ _ 0 = return []
          randCC xs n i = do
            (CC x e) <- oneof [randFor xs (n `div` 2), randIf xs (n `div` 2)]
            if null x
              then do
                fs <- randCC xs n (i-1)
                return $ e++fs
              else do
                fs <- randCC (x:xs) n (i-1)
                return $ e++fs

--custom datatypes in order to create arbitrary programs
```

```haskell
    using the same variable set
data CC = CC String [CClause]
data S = S String Stmt
--since Program is just a type, we have to create a newtype in order to instantiate Arbitrary
newtype P = P Program
  deriving (Eq,Show)

--arbitrary statement
arbitraryS :: [String] -> Gen S
arbitraryS xs = oneof [arbSDef xs, arbSExp xs]
  where
    arbSDef xs = do
      x <- asciiLetter
      e <- arbitraryExp xs
      return $ S [x] $ SDef [x] e
    arbSExp xs = do
      e <- arbitraryExp xs
      return $ S "" $ SExp e

--arbitrary program
instance Arbitrary P where
  arbitrary = sized (\x -> if x == 0 then arb [] 5 else arb [] x)
    where
      arb :: [String] -> Int -> Gen P
      arb _ 0 = return $ P []
      arb xs n = do
        (S x y) <- arbitraryS xs
        if null x
          then do
            (P ys) <- arb xs (n-1)
            return $ P $ y:ys
          else do
            (P ys) <- arb (x:xs) (n-1)
            return $ P $ y:ys

--------------------------print a program as input to parser--------------------------
printProgram :: Program -> String
printProgram [] = ""
printProgram [x] = printStatement x ++ "\n"
printProgram (x:xs) = printStatement x ++ ";\n" ++ printProgram xs

printStatement :: Stmt -> String
```

```haskell
printStatement (SDef v e) = v ++ " = (" ++ printExpression e ++ ")"
printStatement (SExp e) = printExpression e

--print expression: heavy bracketing in order to remove ambiguity
printExpression :: Exp -> String
printExpression (Const v) = printValue v
  where
    printValue NoneVal = "None"
    printValue TrueVal = "True"
    printValue FalseVal = "False"
    printValue (IntVal i) = show i
    printValue (StringVal s) = "'" ++ s ++ "'"
    printValue (ListVal vs) = "[" ++ take (length list - 2) list ++ "]"
      where list = concat ["(" ++ printValue v ++ "), " | v <- vs]
printExpression (Var s) = s
printExpression (Oper o e f) = "((" ++ printExpression e ++ ") " ++ printOperation o ++ " (" ++ printExpression f ++ "))"
  where
    printOperation Plus = "+"
    printOperation Minus = "-"
    printOperation Times = "*"
    printOperation Div = "//"
    printOperation Mod = "%"
    printOperation Eq = "=="
    printOperation Less = "<"
    printOperation Greater = ">"
    printOperation In = "in"
printExpression (Not e) = "(not (" ++ printExpression e ++ "))"
printExpression (Call s es) = "(" ++ s ++ "(" ++ take (length list - 2) list ++ "))"
  where list = concat [printExpression e ++ ", " | e <- es]
printExpression (List es) = "[" ++ take (length list - 2) list ++ "]"
  where list = concat ["(" ++ printExpression e ++ "), " | e <- es]
printExpression (Compr e cs) = "[(" ++ printExpression e ++ ") " ++ take (length list - 1) list ++ "]"
  where
    list = concat [printCClause c ++ " " | c <- cs]
```

```haskell
    printCClause (CCIf e) = "if (" ++ printExpression e ++ ")"
    printCClause (CCFor s e) = "for " ++ s ++ " in (" ++ printExpression e ++ ")"

---------------------------------actual tests---------------------------------

--test parsing of arbitrary programs
propEquality :: P -> Bool
propEquality (P ps) = case parseString (printProgram ps) of
  (Left _) -> trace "fehler" False
  (Right p) -> p == ps


main :: IO ()
main = defaultMain $ localOption (mkTimeout 10000000) tests

tests = testGroup "Tests"
  [
  testProperty "Parse Arbitrary programs and check for equality (may take a second or two)" propEquality,
  testGroup "Specific unit tests" [
    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")
    ],
    testGroup "Tests of skip/comments" [
      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 "skipping comments" $ parseString "True#comment\n#anotherone\t\n;False" @?= (Right [SExp (Const TrueVal),SExp (Const FalseVal)]),
      testCase "skipping comments at eof" $ parseString "True#commentateof" @?= (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"))])
    ],
    testGroup "Tests of pIdent" [
      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 "_cr7 < Messi10_" @?= (Right [SExp (Oper Less (Var "_cr7") (Var "Messi10_"))])
    ],
```

```
    testGroup "Tests of pNumConst" [
      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 '- 911'" $ parseString "- 911" @?= (Left "cannot parse"),
      testCase "leading zeros '007'" $ parseString "007" @?= (Left "cannot parse")
      ],
    testGroup "Tests of pStringConst" [
      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")
      ]
    ]
  ]
```