Advanced Programming
Assignment 3
BOA Parser

GRC196 SZW935

September 29, 2021

For our implementation we chose to use `Parsec`. We are both retaking this class, but our `BoaParser` is completely written from scratch. For the warm-up we did the `Parsec` from scratch and re-used the implementation with `ReadP` we did last year.

1 Completeness

We attempted every section of the specification, with varying degrees of success (see section 2). We heavily used `try` to implement backtracking, often using a symbol at the start or in the middle (see efficiency section) to separate cases. Spacing and comments were dealt with in a `lexeme` function, called in various nonterminals and terminals. We used `lookAhead` frequently, as we did not always know how to check for the end of something. For example, we check `lookAhead (oneOf ")]")` in `exprz`. For `ident` and `stringConst` we attempted two helper functions: `isIdent` and `isStringConst`. They are used with some form of `many` while parsing. `isIdent` checks if there is a keyword followed by `'='` or `'('`, and then fails if so. `isStringConst` tries to convert backslash characters, as well as make sure the characters are printable ascii.

2 Correctness

Our test suite (with 16 out of 94 tests failing) tells us that although we managed to complete much of the specification, we did not manage to complete the following:

- Keywords in `ident` only fail if they are followed by `'='` or `'('`, so `parseString "None"` is a false positive. Also `"not1"` gets recognized as `Not 1`
- Operations are not correctly left-associative or non-associative. Our attempts at fixing this introduced left-recursion to the grammar
- Comments with newline characters do not work

- Backslash characters do not parse correctly in strings.

We spent a long time trying to fix these issues, and think our implementation must be close to working. For the left-recursion, we tried to rewrite the grammar as can be seen in the figure 7.1 in the appendix.

3 Efficiency

Due to our frequent use of `try` our code is not particularly efficient, especially in cases where large parsing is done before having to backtrack. There are also cases where we could have left-factorized the grammar to improve efficiency, for example in `expr` each alternative parses a `expr'` before possibly backtracking. Tests will occasionally time out due to taking longer than `1s`. Our focus was on meeting the specifications first, and due to this we did not have time to refactor the code and produce a more efficient code.

4 Robustness

As in the previous assignment, we want to display errors with a meaningful message. Currently our code will result in errors when it should parse and vice versa when we test it. E.g. as the string `"x+y-z"` parses but in the wrong order or the string `"x*not y"` passes since we aren't properly dealing with operators. As for testing we have tested for simple cases to more complex cases. Simple cases works fairly well. Whereas, when it becomes a more complex string to parse it may not parse.

5 Maintainability

In our current version of the parser the maintainability is on low end. As mentioned, our focus was first to get our code working. We put sequences contained in `do` in one line, making it difficult to read at times. There are probably also places where we could move reused code to an auxiliary function, but again it was not our focus.

6 Other

As mentioned, we had a hard time understanding what it was we didn't understand. It is difficult to know when it is an issue of misunderstanding Parsec, not having the right grammar, or being on the right path but not finding a solution yet. We would appreciate if we could get some helpful comments and be pointed in the correct direction, whether for the re-submission or for the exam.

7 Appendix

7.1 Eliminating Left-Recursion

```
prog    := stmtS

stmtS   := stmt | stmt ";" stmtS'
                              ↑
stmtS'  := stmtS  ← (? not sure)

stmt    := ident "=" expr | expr

expr    := term eopt

eopt    := "+" term eopt | "-" term eopt | ε

term    := factor' topt

topt    := "*" factor' topt | "//" factor' topt | "% factor' topt | ε

factor' := factor fopt
                                              osv
fopt    := "==" factor fopt | "<" factor fopt | ... | ε

factor  := numConst | stringConst | "None" | "True" | "False" |
           ident | "not" expr | "(" expr ")" | ident "(" exprZ ")" |
           "[" exprZ "]" | "[" expr forClause clauseZ "]"

forClause := "for" ident "in" expr

ifClause  := "if" expr
                                              (? not sure)
                                             ↙
clauseZ   := forClause clauseZ | ifClause clauseZ

exprZ     := exprS | ε

exprS     := expr | exprS' ↖

exprS'    := expr "," exprS ε

  ident :=      ⎫
numConst :=     ⎬ see assignment
stringConst :=  ⎭     text
```

7.2 BOA

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

module BoaParser (ParseError, parseString) where

import BoaAST
-- add any other other imports you need

import Text.ParserCombinators.Parsec
import Data.Char

parseString :: String -> Either ParseError Program
parseString = parse parseBoa "parse error"


-- REWRITTEN GRAMMER (or at least, the part that was rewritten)
-- Expr   := Expr' NegOp Expr | Expr' Op Expr | Expr'

-- Expr'  := numConst
--         | stringConst
--         |    None
--         |    True
--         |    False
--         | ident
--         |    not    Expr
--         |    (    Expr    )
--         | ident    (    Exprz    )
--         |    [    Exprz    ]
--         |    [    Expr ForClause Clausez    ]

-- Op     := '+' | '-' | '*' | "//" | '%' | "==" | "<" | ">" | "in"
-- negOp  := "!=" | "<=" | ">=" | "not in"

parseBoa :: Parser Program
parseBoa = do spaces; p <- stmts; eof; return p

stmts :: Parser [Stmt]
stmts = try (do s <- stmt; symbol ";"; ss <- stmts; return (s:ss))
    <|> do s <- stmt; return [s]

stmt :: Parser Stmt
stmt = try (do i <- ident; symbol "="; e <- expr; return $ SDef i e)
    <|> do e <- expr; return (SExp e)

expr :: Parser Exp
expr = lexeme $ try (do e <- expr'; o <- negOp; e' <- expr; return $ Not $ Oper o e e')
    <|> try (do e <- expr'; o <- op; e' <- expr; return $ Oper o e e')
    <|> do expr'

expr' :: Parser Exp
expr' = lexeme $ do n <- numConst; return (Const (IntVal n))
    <|> do symbol "\'"; s <- stringConst; return (Const (StringVal s))
    <|> do symbol "None"; return (Const NoneVal)
    <|> do symbol "True"; return (Const TrueVal)
    <|> do symbol "False"; return (Const FalseVal)
    <|> try (do string "not"; spaces; e <- expr; return (Not e))
    <|> try (do i <- ident; symbol "("; es <- exprz; symbol ")"; return (Call i es))
    <|> do i <- ident; return (Var i)
    <|> do symbol "("; e <- expr; symbol ")"; return e
    <|> try (do symbol "["; es <- exprz; symbol "]"; return (List es))
    <|> do symbol "["; e <- expr; fc <- forClause; cs <- clausez; symbol "]"; return (Compr e (fc:cs))

op :: Parser Op
op = lexeme $ do char ’+’; return Plus
    <|> do char ’-’; return Minus
    <|> do char ’*’; return Times
    <|> do string "//"; return Div
    <|> do char ’%’; return Mod
    <|> do string "=="; return Eq;
    <|> do char ’<’; return Less
    <|> do char ’>’; return Greater
    <|> do string "in"; requiredSpaces; return In

negOp :: Parser Op
negOp = lexeme $ do string "!="; return Eq
    <|> do string "<="; return Greater
    <|> do string ">="; return Less
    <|> do string "not in"; return In

forClause :: Parser CClause
forClause = do string "for"; requiredSpaces; i <- ident; string "in"; requiredSpaces; e <- expr; return (CCFor i e)

ifClause :: Parser CClause
ifClause = do string "if"; requiredSpaces; e <- expr; spaces; return (CCIf e)

requiredSpaces :: Parser ()
requiredSpaces = do lookAhead (satisfy isSpace); spaces; return ()

clausez :: Parser [CClause]
clausez = lexeme $ try (do lookAhead (char ’]’); return [])
    <|> do fc <- forClause; cs <- clausez; return (fc:cs)
    <|> do ic <- ifClause; cs <- clausez; return (ic:cs)

exprz :: Parser [Exp]
exprz = try (do lookAhead (oneOf ")]"); return [])
    <|> do exprs

exprs :: Parser [Exp]
exprs = try (do e <- expr; symbol ","; es <- exprs; return (e:es))
    <|> do e <- expr; return [e]

ident :: Parser String
ident = lexeme $ do lookAhead (noneOf "0123456789"); many1 isIdent;

isIdent :: Parser Char
isIdent = (isKeyword >> fail "reserved keyword")
    <|> satisfy (\c -> isAlphaNum c || c == ’_’)

isKeyword :: Parser ()
```

```haskell
isKeyword = try (do string "None"; spaces; oneOf "=("; return ())
    <|> try (do string "True"; spaces; oneOf "=("; return ())
    <|> try (do string "False"; spaces; oneOf "=("; return ())
    <|> try (do string "for"; spaces; oneOf "=("; return ())
    <|> try (do string "if"; spaces; oneOf "=("; return ())
    <|> try (do string "in"; spaces; oneOf "=("; return ())
    <|> try (do string "not"; spaces; oneOf "=("; return ())

numConst :: Parser Int
numConst = lexeme $ do string "-"; ds <- getDigits; return (read ("-"++ds))
    <|> do ds <- getDigits; return (read ds)

getDigits :: Parser String
getDigits = do char '0'; return "0"
    <|> do lookAhead (noneOf "0"); many1 digit;

stringConst :: Parser String
stringConst = do manyTill isStringConst (char '\'')

isStringConst :: Parser Char
isStringConst = newline
    <|> (string "\\\\" >> return '\\')
    <|> (string "\\\'" >> return '\'')
    <|> (char '\\' >> fail "single backslash")
    <|> do lookAhead $ satisfy isPrint; satisfy isAscii

comment :: Parser ()
comment = try (do char '#'; skipMany (noneOf "\\"); char '\n'; return ())
    <|> do char '#'; skipMany (noneOf "\\"); eof

lexeme :: Parser a -> Parser a
lexeme p = try (do a <- p; comment; return a)
    <|> do a <- p; spaces; return a

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

Listing 1: BoaInterp.hs

7.3 Test of BOA

```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 "Parser 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]

      -- space tests
      testCase "space at start" $
        parseString " x" @?=
          Right [SExp (Var "x")],
      testCase "tab at start" $
        parseString "\tx" @?=
          Right [SExp (Var "x")],
      testCase "newline at start" $
        parseString "\nx" @?=
          Right [SExp (Var "x")],

      -- numConst tests
      testCase "1" $
        parseString "1" @?=
          Right [SExp (Const (IntVal 1))],
      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 "int overflow" $
        parseString "18446744073709551615" @?=
          Right [SExp (Const (IntVal (-1)))],
      testCase "00" $
        case parseString "00" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "007" $
        case parseString "007" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "- 1" $
        case parseString "- 1" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "+1" $
        case parseString "+1" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,

      -- ident tests
      testCase "x" $
        parseString "x" @?=
          Right [SExp (Var "x")],
      testCase "x=1" $
        parseString "x=1" @?=
          Right [SDef "x" (Const (IntVal 1))],
      testCase "x = 1" $
        parseString "x = 1" @?=
          Right [SDef "x" (Const (IntVal 1))],
      testCase "var" $
        parseString "var" @?=
          Right [SExp (Var "var")],
      testCase "not1" $
        parseString "not1" @?=
          Right [SExp (Var "not1")],
      testCase "false1" $
        parseString "false1" @?=
          Right [SExp (Var "false1")],
      testCase "var_1" $
        parseString "var_1" @?=
          Right [SExp (Var "var_1")],
      testCase "_var1" $
        parseString "_var1" @?=
          Right [SExp (Var "_var1")],
      testCase "1_var" $
        case parseString "1_var" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      --reserved words in ident
      testCase "None=1" $
        case parseString "None=1" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "True = 1" $
        case parseString "True = 1" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase " False =   1" $
        case parseString " False =   1" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "for" $
        case parseString "for" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "if()" $
```

```
        case parseString "if()" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "in=1" $
        case parseString "in=1" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "not=1" $
        case parseString "not=1" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,

      -- stringConst tests
      testCase "'a'" $
        parseString "\'a\'" @?=
          Right [SExp (Const (StringVal "a"))],
      testCase "'hello world'" $
        parseString "\'hello world\'" @?=
          Right [SExp (Const (StringVal "hello world"))],
      testCase "'!\"#$%&()*+,-./:;<=>?[]^_@`}{|~'" $ --TODO: ascii delete is
    printable?
        parseString "'!\"#$%&()*+,-./:;<=>?[]^_@`}{|~'" @?=
          Right [SExp (Const (StringVal"!\"#$%&()*+,-./:;<=>?[]^_@`}{|~"))],
      testCase "'x\\y\\z'" $
        parseString "'x\\\\y\\\\z'" @?=
          Right [SExp (Const (StringVal "x\\y\\z"))],
      testCase "'x\\y\'z'" $
        parseString "'x\\\\y\'z'" @?=
          Right [SExp (Const (StringVal "x\\y'z"))],
      testCase "'a\\nb'" $
        parseString "'a\\nb'" @?=
          Right [SExp (Const (StringVal "a\nb"))],
      testCase "'fo\\\\o\\b\\na\\\'r'" $
        parseString "'fo\\\\o\\b\\na\\\'r'" @?=
          Right [SExp (Const (StringVal "fo\\ob\na'r"))],
      testCase "'a\"b\\n'" $
        parseString "'a\"b\\n'" @?=
          Right [SExp (Const (StringVal "a\"b\\n"))],
      testCase "'\\n'" $
        parseString "'\n'" @?=
          Right [SExp (Const (StringVal "\n"))],
      testCase "'  '" $
        parseString "'  '" @?=
          Right [SExp (Const (StringVal "\235"))],
      testCase "'\\t'" $
        case parseString "'\t'" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "'\\DEL'" $
        case parseString "'\DEL'" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "'\\'" $
        case parseString "'\\'" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,
      testCase "'\''" $
        case parseString "'\''" of
          Left e -> return ()
          Right p -> assertFailure $ "Unexpected parse: " ++ show p,

      -- other expr tests
      testCase "NoneVal" $
        parseString "None" @?=
          Right [SExp (Const NoneVal)],
      testCase "TrueVal" $
        parseString "True " @?=
          Right [SExp (Const TrueVal)],
      testCase "FalseVal" $
        parseString "False" @?=
          Right [SExp (Const FalseVal)],
          -- oper tested comprehensively below
      testCase "not 1" $
        parseString "not 1" @?=
          Right [SExp (Not (Const (IntVal 1)))],
      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))))],
      testCase "(1)" $
        parseString "(1)" @?=
          Right [SExp (Const (IntVal 1))],
      testCase "(((1)))" $
        parseString "(((1)))" @?=
          Right [SExp (Const (IntVal 1))],
      testCase "( 1 )" $
        parseString "( 1 )" @?=
          Right [SExp (Const (IntVal 1))],
      testCase "f()" $
        parseString "f()" @?=
          Right [SExp (Call "f" [])],
      testCase "f(x)" $
        parseString "f(x)" @?=
          Right [SExp (Call "f" [Var "x"])],
      testCase "f ( x < 4 ) " $
        parseString "f ( x < 4 ) " @?=
          Right [SExp (Call "f" [Oper Less (Var "x") (Const (IntVal 4))])],
      testCase "[]" $
        parseString "[]" @?=
          Right [SExp (List [])],
      testCase "[1]" $
        parseString "[1]" @?=
          Right [SExp (List [Const (IntVal 1)])],
      testCase "[1,2]" $
        parseString "[1,2]" @?=
          Right [SExp (List [Const (IntVal 1),Const (IntVal 2)])],
      testCase "[1, 2, 3]" $
        parseString "[1,2,3]" @?=
          Right [SExp (List [Const (IntVal 1),Const (IntVal 2),Const (IntVal 3)])],
      testCase "[(1+2), True, 'yes']" $
        parseString "[(1+2), True, 'yes']" @?=
```

```haskell
        Right [SExp (List [Oper Plus (Const (IntVal 1)) (Const (IntVal 2)), Const (TrueVal), Const (StringVal "yes")])],
    testCase "[x!=y,x>=y,x<=y,x not in y]" $
      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"))])],

    -- oper tests
    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)))],
    testCase "1% 1" $
      parseString "1% 1" @?=
        Right [SExp (Oper Mod (Const (IntVal 1)) (Const (IntVal 1)))],
    testCase "1 == True" $
      parseString "1 == True" @?=
        Right [SExp (Oper Eq (Const (IntVal 1)) (Const TrueVal))],
    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 "1in [1]" $
      parseString "1in [1]" @?=
        Right [SExp (Oper In (Const (IntVal 1)) (List [Const (IntVal 1)]))],
    testCase "1 not in [1]" $
      parseString "1 not in [1]" @?=
        Right [SExp (Not (Oper In (Const (IntVal 1)) (List [Const (IntVal 1)])))],

    -- clausez tests
    testCase "[ 1for x in x ]" $
      parseString "[ 1for x in x ]" @?=
        Right [SExp (Compr (Const (IntVal 1)) [CCFor "x" (Var "x")])],
    testCase "[ 1 for x in x if 1 == 1]" $
      parseString "[ 1 for x in x if t == 1]" @?=
        Right [SExp (Compr (Const (IntVal 1)) [CCFor "x" (Var "x"), CCIf (Oper Eq (Var "t") (Const (IntVal 1)))])],
    testCase "[if x]" $
      case parseString "[if x]" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,

    -- keyword space tests
    testCase "2 in[1]" $
      case parseString "2 in[1]" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
    testCase "[1 forx in x]" $
      case parseString "[1 forx in x]" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
    testCase "[1 for x inx]" $
      case parseString "[1 for x inx]" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,
    testCase "[1for x in x ifx]" $
      case parseString "[1for x in x ifx]" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,

    -- associativity tests
    testCase "1-2+3" $
      parseString "1-2+3" @?=
        Right [SExp (Oper Plus (Oper Minus (Const (IntVal 1)) (Const (IntVal 2))) (Const (IntVal 3)))],
    testCase "(1-2)+3" $
      parseString "(1-2)+3" @?=
        Right [SExp (Oper Plus (Oper Minus (Const (IntVal 1)) (Const (IntVal 2))) (Const (IntVal 3)))],
    testCase "1-(2+3)" $
      parseString "1-(2+3)" @?=
        Right [SExp (Oper Minus (Const (IntVal 1)) (Oper Plus (Const (IntVal 2)) (Const (IntVal 3))))],
    testCase "1*2//3" $
      parseString "1*2//3" @?=
        Right [SExp (Oper Div (Oper Times (Const (IntVal 1)) (Const (IntVal 2))) (Const (IntVal 3)))],
    testCase "(1*2)//3" $
      parseString "(1*2)//3" @?=
        Right [SExp (Oper Div (Oper Times (Const (IntVal 1)) (Const (IntVal 2))) (Const (IntVal 3)))],
    testCase "1*(2//3)" $
      parseString "1*(2//3)" @?=
        Right [SExp (Oper Times (Const (IntVal 1)) (Oper Div (Const (IntVal 2)) (Const (IntVal 3))))],
    testCase "x<y<z" $
      case parseString "x<y<z" of
        Left e -> return ()
        Right p -> assertFailure $ "Unexpected parse: " ++ show p,

    -- statement tests
    testCase "x; x=1;1" $
```

```
        parseString "x; x=1;1" @?=
          Right [SExp (Var "x"), SDef "x" (Const (IntVal 1)), SExp (Const (IntVal 1))],

      -- comment tests
      testCase "x#bar" $
        parseString "x#bar" @?=
          Right [SExp (Var "x")],
      testCase "x#bar\\n" $
        parseString "x#bar\\n" @?=
          Right [SExp (Var "x")],
      testCase "x#\\n" $
        parseString "x#\\n" @?=
          Right [SExp (Var "x")],
      testCase "x#\\n=1" $
        parseString "x#\\nbar" @?=
          Right [SDef "x" (Const (IntVal 1))],
      testCase "x#bar 78 \\n = 1" $
        parseString "x#bar 78 \\n = 1" @?=
          Right [SDef "x" (Const (IntVal 1))]
    ]
```

Listing 2: Test.hs