AP assignment 2: Boa Parser

sortraev, wlc376

2020-09-23

0 Design and Implementation

In this section, I present and explain key design choices, and features of my implementation; choice of parser library, disambiguation of grammar, handling of whitespace, and limitations.

The handed-in code in `code/part2/src` (see appendix A) is supplied with simple code comments where appropriate.

0.1 Choice of parser library; consequences thereof

I have chosen `ReadP` for my implementation, and did so for the simple reason that I was given the impression that it would make for a breezier implementation.

As of yet I have not looked much into the `Parsec` library, but I am confident that this has been true; the `ReadP` is far simpler, yet sufficient for the task at hand, so I am mostly satisfied with my choice.

However, the `ReadP` library does not support particularly helpful error handling. In fact, only two types of errors are handled; parsing failures and ambiguous parses. In the future, it might be advantageous for me to port my parser to `Parsec`.

0.2 Disambiguating the Boa grammar

The Boa grammar, as presented in the assignment text, is highly ambiguous with respect to parsing of the `Expr` non-terminal. The assignment text defines associativity and precedence of both unary and binary operators; before implementation into `ReadP`, I need to eliminate left-recursion and ambiguity in the grammar.

To do this, I left-factor and re-structure the grammar into a parsing tree with recursive descent. Disambiguation is only necessary for the *Expr* non-terminal, so I only apply modifications here. A snippet of the relevant changes to the grammar is presented in table 1 below:

Exp	:=	*NotExp*
NotExp	:=	‘`not`‘ *Exp* | *RelBinOpExp*
RelBinOpExp	:=	*ArithBinOpExp1* *RelBinOp* *ArithBinOpExp1*
	|	*ArithBinOpExp1*
ArithBinOpExp1	:=	*ArithBinOpExp2* *ArithBinOpExp1’*
ArithBinOpExp1’	:=	*ArithBinOp1* *ArithBinOpExp2* *ArithBinOpExp1’*
	|	ε
ArithBinOpExp2	:=	*Atom ArithBinOpExp2’*
ArithBinOpExp2’	:=	*ArithBinOpExp2* *Atom* *ArithBinOpExp2’*
	|	ε
RelBinOp	:=	‘`<`‘ | ‘`<=`‘ | ‘`>`‘ | ‘`>=`‘
	|	‘`==`‘ | ‘`!=`‘ | ‘`in`‘ | ‘`not in`‘
ArithBinOp1	:=	‘`+`‘ | ‘`-`‘
ArithBinOp2	:=	‘`*`‘ | ‘`//`‘ | ‘`\%`‘
Atom	:=	**numConst**
	|	**stringConst**
	|	**ident**
	|	‘`None`‘ | ‘`True`‘ | ‘`False`‘
	|	‘`[`‘ *Exprz* ‘`]`‘
	...	

Table 1: Snippet of revised Boa grammar.

0.2.1 Implementation of disambiguated grammar

Below snippet shows how parsing of the disambiguated *Expr* is implemented using `ReadP`:

```
1  parseExp :: ReadP Exp
2  parseExp = parseNotExp
3
4  parseNotExp :: ReadP Exp
5  parseNotExp = (keywordNot >> Not <$> parseExp) <|> parseRelBinOps
6
7  parseRelBinOps :: ReadP Exp
8  parseRelBinOps = (parseArithBinOps >>= relBinOp) <*> parseArithBinOps
9    where relBinOp e = termLess      $> Oper Less          e
10                        ...
11                   <|> termGreaterEq $> Not . Oper Less    e
12
13 parseArithBinOps :: ReadP Exp
14 parseArithBinOps = infix1
15   where infix1 = chainl1 infix2    $ Oper <$> (termPlus  $> Plus
16                                           <|> ...
17         infix2 = chainl1 parseAtom $ Oper <$> (termTimes $> Times
18                                           <|> ...
19 parseAtom :: ReadP Exp
20 parseAtom = lexeme $ parseConst
21                  <|> ...
22                  <|> parenthesized parseExp
```

The non-associative relational operators are parsed quite straightforwardly with a direct translation from grammar to a `ReadP` parser function in lines 7-11.

Left-associative parsing of the *ParseArithBinOps* non-terminal of the grammar is also handled with a breeze using the `ReadP`-built-in `chainl1` function (lines 15, 17); this function performs a left-fold over a sequence of zero or more binary operations, and as such it also parses atom tokens.

Alas, there exists a prettier solution that is more consistent in the handling of relational and arithmetic operators, but this solution is effective nonetheless.

0.3 Handling whitespace in parsing

Here would be a more fleshed-out section on the handling of whitespace, but apparently the deadline is 20:00 and not midnight. Dang.

In any case, I largely follow Filinski's advice from one of his parsing lectures, which in broad terms is to simply prepend a `skipSpaces` to (most) all parsers.

0.3.1 Whitespace: keywords and identifiers

However, wrt. keywords and identifiers, the problem of whitespace becomes more subtle; here, we have to assert that what follows is not alphanumeric or underscores, since these are valid keyword/identifier characters.

My solution is to simply eat up as much as possible, perform a single character look-ahead and assert that this is a legal follow-up to a keyword/identifier. I do so with the following helper functions (from `BoaParser.Extras`):

```haskell
-- can this string immediately follow a keyword?
-- also used by `parseIdent`.
canFollowKeyword :: String -> Bool
canFollowKeyword (c:_) = not (isDigit c || isAlpha c || c == '_')
canFollowKeyword _     = True
```

This function is then used in conjunction with identifier and keyword parsing. Below is a snippet of its usage in `ParseExp.parseIdent`:

```haskell
parseIdent :: ReadP String
parseIdent = ident >>= \i -> look >>= guard . canFollowKeyword
  >> guard (i `notElem` reservedKeywords) >> return i

  where ident     = prepend identHead identTail
        identHead = letter <|> char '_'
        identTail = many (identHead <|> num)
```

0.3.2 Whitespace: code comments and tokenization

Below is my implementation of a parser which skips an arbitrary number of code comments with an arbitrary amount of leading and trailing whitespace, and the helper function I use for tokenizing parsers. The code comments should speak for themselves.

```haskell
-- skip zero or more comments, where a comment is:
--   * a leading '#' preceded by zero or more whitespace characters.
--   * zero or more arbitrary characters (the assignment text does not specify
--       restrictions on the contents of comments).
--   * a terminating newline or EOF.
skipComment :: ReadP ()
skipComment = many (skipSpaces >> char '#' >>
                manyTill (satisfy isPrint) parseCommentTerminator) >>
                 skipSpaces
 where parseCommentTerminator = eof <|> satisfy (== '\n') $> ()

-- used to tokenize parsers. skips leading comments and/or whitespace.
lexeme :: ReadP a -> ReadP a
lexeme p = skipComment >> p
```

`skipComment` could probably do with some optimization; in particular, this is probably not very efficient (or well-thought-out) usage of `skipSpaces`, but it is effective.

0.4 Known limitations and grammar redefinitions

Before validation testing, my implementation has only one known limitation - emphasis on *known*; I do not claim that my program does not have *many* more limitations, but we shall remain agnostic about these until validation testing.

0.4.1 Limitation: Boa string literal parsing

In designing and implementing parsing of string literals, I have decided to make an arguably *radical* simplification of the Boa grammar. A Boa string literal is now: **one opening single-quote, followed by zero or more printable characters or whitespace, terminated by one closing single-quote**. In this regard, a 'printable character' is any char which satisfies `isPrint`, and whitespace is any char which satisfies `isSpace` (both from `Data.Char`.

As such, Boa string literal parsing simply becomes:

```
-- TODO: further restrictions :(
parseStringVal :: ReadP Value
parseStringVal = StringVal <$> between (char '\'') (char '\'')
                                (many (satisfy isString)
  where isString c = isPrint c || isSpace c
```

This choice was essentially a trade-off between satisfying specifications and handing in a passable, tested product by deadline.

1 Testing

In this section, I discuss my validation test plan and report the results of testing.

1.1 Reproduction of tests

I use `Tasty` for testing. All of my tests can be viewed in `code/part2/tests` or appendix B to this report.

To reproduce tests, navigate to `code/part2` and run `stack test`.

1.2 Testing goals

The goal of the test plan is to attain full edge case coverage with unit testing of each implemented function and helper function. I also ideally want to test each possible type of parsing failure (recall from section 0.1 that the `ReadP` library does not support overly telling error messages; this restricts negative testing to a simple “success/fail“ distinction).

1.3 Test plan

1.3.1 Test plan: strategy

Filinski has suggested on the absalon discussion forum that the program can be satisfiably tested using simply the top-level `BoaParser.parseString` interface, since all expressions are programs in their own right, and as such it should be possible to essentially unit test individual functionalities using a white-box testing strategy, which is what I'll aim at.

I want to test correct handling and parsing of each language feature, and thus to touch upon each constructor in the AST. In addition, I want to test correct handling of whitespace, both where it is and is not required.

Wrt. negative testing, for each constructor, I want to test various types of parsing failures (eg. association or whitespacing errors).

1.3.2 Testing: test suite

For lack of time, I won't go into detail with my test suite, but each test has a fitting name explaining just what that test asserts, so I will advice the reader to view the test cases in my test plan by running `stack test`, or, alternatively, in the source code in `code/part2/tests/BoaTests`.

1.4 Validation testing results

All of my own validation tests pass successfully.

All of OnlineTA's validation tests pass successfully, **save for most of the test cases involving Boa string literals** (which were not expected to pass; see section 0.4).

1.5 Evaluation

My final test suite includes 128 tests, and I manage to cover mostly every unit test case I had planned for.

Based on validation test results alone, I am almost convinced that my implementation is sound; I would have liked to have rigorously sought out possibly uncovered edge cases, but once again, I succumb to the deadline.

More importantly, I have not performed any actual integration tests, so I cannot say for certain that my implementation would not break on general inputs (eg. large, contrived programs).

Aside from validation test results, I am generally satisfied with my implementation and what it has taught me of parser combinators and Haskell programming in general.

Appendix

A Code: Implementation

A.1 code/part2/src/BoaParser.hs

```
module BoaParser (ParseError, parseString) where

import BoaAST
import Text.ParserCombinators.ReadP

import BoaParser.ParseStmt

type ParseError = String

parseString :: String -> Either ParseError Program
parseString str =
  case readP_to_S (parseProgram <* eof) str of
    [(exp, _)] -> Right exp
    []         -> Left $ "No parse of: " ++ str
    out@((_, _):_rest) -> Left $ "Ambiguous parse. Unparsed: "
                               ++ show' out

show' :: [(a, String)] -> String
show' [] = ""
show' ((_, b):rest) = (if not (null b) then b ++ ", " else "") ++ show' rest
```

A.2 code/part2/src/BoaParser/ParseExp.hs

```
module BoaParser.ParseExp where

import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))
import Data.Functor (($>))
import Data.Char (isPrint, isSpace)
import Control.Monad (guard)

import BoaAST
import BoaParser.Extras

---------------------
---- Exp parsing ----
---------------------
parseExp :: ReadP Exp
parseExp = parseNotExp

parseNotExp :: ReadP Exp
parseNotExp = (keywordNot >> Not <$> parseExp) <|> parseRelBinOps
```

```haskell

parseRelBinOps :: ReadP Exp
parseRelBinOps = (parseArithBinOps >>= relBinOp) <*> parseArithBinOps
                 <|> parseArithBinOps
  where relBinOp e = (termLess      $> Oper Less          e)
                 <|> (termGreater   $> Oper Greater       e)
                 <|> (termEq        $> Oper Eq            e)
                 <|> (keywordIn     $> Oper In            e)
                 <|> (termGreaterEq $> Not . Oper Less    e)
                 <|> (termLessEq    $> Not . Oper Greater e)
                 <|> (termNotEq     $> Not . Oper Eq      e)
                 <|> (keywordNotIn  $> Not . Oper In      e)

parseArithBinOps :: ReadP Exp
parseArithBinOps = infix1
  where infix1 = chainl1 infix2    $ Oper <$> (termPlus  $> Plus
                                          <|>  termMinus $> Minus)
        infix2 = chainl1 parseAtom $ Oper <$> (termTimes $> Times
                                          <|>  termDiv   $> Div
                                          <|>  termMod   $> Mod)
parseAtom :: ReadP Exp
parseAtom = lexeme $ parseConst
                 <|> parseVarExp
                 <|> parseComprExp
                 <|> parseListExp
                 <|> parseCallExp
                 <|> parenthesized parseExp

parseVarExp :: ReadP Exp
parseVarExp = Var <$> parseIdent


---------------------------------
---- Const and value parsing ----
---------------------------------
parseConst :: ReadP Exp
parseConst = Const <$> (parseIntVal <|> parseStringVal <|> parseMiscVal)

parseListExp :: ReadP Exp
parseListExp  = bracketed $ List <$> sepBy parseExp listDelim

parseComprExp :: ReadP Exp
parseComprExp = bracketed $ Compr <$> parseExp <*> cclauses

  where cclauses = ccFor `prepend` many cclause
        cclause  = ccFor <|> ccIf
        ccFor    = keywordFor >> CCFor <$> (parseIdent' <* keywordIn) <*> parseExp
        ccIf     = keywordIf  >> CCIf  <$> parseExp


parseCallExp :: ReadP Exp
parseCallExp = Call <$> parseIdent <*> parenthesized parseArgs
  where parseArgs = sepBy parseExp listDelim


-----------------------
---- Value parsing ----
-----------------------
parseIntVal :: ReadP Value
parseIntVal = IntVal <$> (parsePosInt <|> parseNegInt)
  where parsePosInt =          read <$> digits
        parseNegInt = negate . read <$> ((char '-') >> digits)
        digits = (string "0") <|> (positiveNum `prepend` many num)

-- TODO: restrictions on string literals; see assignment text
```

```haskell
parseStringVal :: ReadP Value
parseStringVal = StringVal <$> between (char '\'') (char '\'')
                               (many (satisfy isString))
  where isString c = isPrint c || isSpace c


parseMiscVal :: ReadP Value
parseMiscVal = (termTrue  $> TrueVal)
           <|> (termFalse $> FalseVal)
           <|> (termNone  $> NoneVal)

--------------
---- misc ----
--------------
parseIdent, parseIdent' :: ReadP String
parseIdent = ident >>= \i -> look >>= guard . canFollowKeyword
  >> guard (i `notElem` reservedKeywords) >> return i

  where ident     = prepend identHead identTail
        identHead = letter <|> char '_'
        identTail = many (identHead <|> num)

parseIdent' = lexeme $ parseIdent
```

A.3 code/part2/src/BoaParser/ParseStmt.hs

```haskell
module BoaParser.ParseStmt where

import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))

import BoaAST
import BoaParser.Extras
import BoaParser.ParseExp


-- a Program is:
--   * 1 or more semicolon-separated statements (the last statement should
--       not have a trailing semicolon)
--   * an arbitrary amount of trailing comments/whitespace.
parseProgram :: ReadP Program
parseProgram = sepBy1 parseStmt termSemicolon <* skipComment


----------------------
---- Stmt parsing ----
----------------------
parseStmt :: ReadP Stmt
parseStmt = parseStmtDef <|> parseStmtExp

parseStmtDef :: ReadP Stmt
parseStmtDef = SDef <$> (parseIdent' <* termEquals) <*> parseExp

parseStmtExp :: ReadP Stmt
parseStmtExp = SExp <$> parseExp
```

A.4 code/part2/src/BoaParser/Extras.hs

```haskell
module BoaParser.Extras where

import Text.ParserCombinators.ReadP
import Control.Applicative ((<|>))
import Data.Functor (($>))
import Control.Monad (guard)
import Data.Char


----------------------------------------
---- utilities and helper functions ----
----------------------------------------
prepend :: ReadP a -> ReadP [a] -> ReadP [a]
prepend = (<*>) . (<$>) (:)

anyChar = satisfy (const True)

oneOf :: [Char] -> ReadP Char
oneOf xs = satisfy (`elem` xs)

parenthesized, bracketed :: ReadP a -> ReadP a
parenthesized = between termOpenPar     termClosePar
bracketed     = between termOpenBracket termCloseBracket


-- can this string immediately follow a keyword?
-- also used by `parseIdent`.
canFollowKeyword :: String -> Bool
canFollowKeyword (c:_) = not (isDigit c || isAlpha c || c == '_')
canFollowKeyword _     = True


-- parses a keyword, but only if that keyword is followed by
-- by something that can legally follow a keyword.
keyword :: String -> ReadP String
keyword kw = string' kw >>= \s -> look >>=
               guard . canFollowKeyword >> return s


-- skip zero or more comments, where a comment is:
--    * a leading '#' preceded by zero or more whitespace characters.
--    * zero or more arbitrary characters (the assignment text does not specify
--        restrictions on the contents of comments).
--    * a terminating newline or EOF.
skipComment :: ReadP ()
skipComment = many (skipSpaces >> char '#' >>
                manyTill (satisfy isPrint) parseCommentTerminator) >>
                 skipSpaces
 where parseCommentTerminator = eof <|> satisfy (== '\n') $> ()


-- used to tokenize parsers. skips and leading comments and/or whitespace.
lexeme :: ReadP a -> ReadP a
lexeme p = skipComment >> p

char' :: Char -> ReadP Char
char'   = lexeme . char
string' :: String -> ReadP String
string' = lexeme . string
```

```haskell

--------------------------------
---- keywords and terminals ----
--------------------------------
letter, num, positiveNum, listDelim :: ReadP Char
letter  = oneOf (['a'..'z'] ++ ['A'..'Z'])
num         = oneOf ['0'..'9']
positiveNum = oneOf ['1'..'9']

listDelim   = char' ',' -- TODO: just char ','? whitespace surrounding
                        -- delims might be handled in other parsers.

keywordFor   = keyword "for"
keywordIn    = keyword "in"
keywordIf    = keyword "if"
keywordNot   = keyword "not"
keywordNotIn = keywordNot >> keywordIn

termTrue  = string' "True"
termFalse = string' "False"
termNone  = string' "None"

-- binary op terminals
termPlus  = char' '+'
termMinus = char' '-'
termTimes = char' '*'
termMod   = char' '%'
termDiv   = string' "//"

termLess      = char'   '<'
termLessEq    = string' "<="
termGreater   = char'   '>'
termGreaterEq = string' ">="
termEq        = string' "=="
termNotEq     = string' "!="

-- brackets and parentheses
termOpenPar      = char' '('
termClosePar     = char' ')'
termOpenBracket  = char' '['
termCloseBracket = char' ']'

-- for SDef's
termEquals    = char' '='
termSemicolon = char' ';'

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

B Code: Testing

B.1 code/part2/tests/Test.hs

```haskell
import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaParser

import BoaTests.ConstTests
import BoaTests.ExpTests
import BoaTests.MiscTests

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

allTests :: TestTree
allTests = testGroup "My BoaParser unit tests"
  [
  constTests, expTests, progTests, commentTests
  ]
```

B.2 code/part2/tests/BoaTests/Util.hs

```haskell
module BoaTests.Util where

import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaParser

test :: String -> String -> Program -> TestTree
test testName input expectedOut =
  testCase testName $ parseString input @?= (Right expectedOut)

negTest :: String -> String -> TestTree
negTest testName input = testCase testName $
  case parseString input of
    Left  _ -> return ()
    Right p -> assertFailure $ "Unexpected parse: " ++ show p
```

B.3 code/part2/tests/BoaTests/ConstTests.hs

```haskell
module BoaTests.ConstTests where

import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaTests.Util

constTests :: TestTree
constTests = testGroup ">>>> Const tests"
  [noneTests, trueTests, falseTests, intTests, strTests]


noneTests = testGroup ">> None const tests"
  [noneTest0, noneTest1, noneTest2, noneTest3, noneTest4, noneTest5, noneTest6]
trueTests = testGroup ">> True const tests"
  [trueTest0, trueTest1, trueTest2, trueTest3, trueTest4, trueTest5, trueTest6]
falseTests = testGroup ">> False const tests"
  [falseTest0, falseTest1, falseTest2, falseTest3, falseTest4, falseTest5, falseTest6]
intTests = testGroup ">> Int const tests"
  [intTest0, intTest1, intTest2, intTest3, intTest4, intTest5, intTest6]
strTests = testGroup ">> String literal tests"
  [strTest0, strTest1, strTest2, strTest3, strTest4, strTest5, strTest6, strTest7, strTest8]


noneTest0 = test "None"                     "None"  [SExp (Const NoneVal)]
noneTest1 = test "Trailing garbage"         "Nonex" [SExp (Var "Nonex")]
noneTest2 = test "Leading garbage"          "xNone" [SExp (Var "xNone")]
noneTest3 = test "Leading whitespace"       " None" [SExp (Const NoneVal)]
noneTest4 = test "Trailing whitespace"      "None " [SExp (Const NoneVal)]
noneTest5 = test "Missing capitalization"   "none"  [SExp (Var "none")]
noneTest6 = negTest "Whitespace in keyword" "No ne"


trueTest0 = test "True"                     "True"    [SExp (Const TrueVal)]
trueTest1 = test "Trailing garbage"         "Truex"   [SExp (Var "Truex")]
trueTest2 = test "Leading garbage"          "xTrue"   [SExp (Var "xTrue")]
trueTest3 = test "Leading whitespace"       " True"   [SExp (Const TrueVal)]
trueTest4 = test "Trailing whitespace"      "True "   [SExp (Const TrueVal)]
trueTest5 = test "Missing capitalization"   "true"    [SExp (Var "true")]
trueTest6 = negTest "Whitespace in keyword" "Tr ue"


falseTest0 = test "False"                    "False"  [SExp (Const FalseVal)]
falseTest1 = test "Trailing garbage"         "Falsex" [SExp (Var "Falsex")]
falseTest2 = test "Leading garbage"          "xFalse" [SExp (Var "xFalse")]
falseTest3 = test "Leading whitespace"       " False" [SExp (Const FalseVal)]
falseTest4 = test "Trailing whitespace"      "False " [SExp (Const FalseVal)]
falseTest5 = test "Missing capitalization"   "false"  [SExp (Var "false")]
falseTest6 = negTest "Whitespace in keyword" "Fal se"


intTest0 = test "Just a number"       "43"   [SExp (Const (IntVal 43))]
intTest1 = test "Leading whitespace"  " 43"  [SExp (Const (IntVal 43))]
intTest2 = test "Trailing whitespace" "43 "  [SExp (Const (IntVal 43))]
intTest3 = test "Whitespace"          " 43 " [SExp (Const (IntVal 43))]
intTest4 = test "Neg number"          "-43"  [SExp (Const (IntVal (-43)))]
intTest5 = test "Legal space before negation" " -43"  [SExp (Const (IntVal (-43)))]
intTest6 = negTest "Illegal space after negation" "- 43"

```

```
strTest0 = test "simple string" "'foo'" [SExp (Const (StringVal "foo"))]
strTest1 = test "empty string"  "''"   [SExp (Const (StringVal ""))]
strTest2 = test "comment in string" "'fooabc#commentinstring'"
             [SExp (Const (StringVal "fooabc#commentinstring"))]
strTest3 = test "Test non-vanilla Boa chars" "'\\\n'" [SExp (Const (StringVal "\\\n"))]
strTest4 = test "Non-vanilla Boa single-quote " "'foo''" [SExp (Const (StringVal "foo'"))]
strTest5 = negTest "Uon-printable characters" "'\DEL'"
strTest6 = test "Whitespace before" "\t   'foo'" [SExp (Const (StringVal "foo"))]
strTest7 = test "Whitespace after" "'foo'  \n  " [SExp (Const (StringVal "foo"))]
strTest8 = negTest "Uore non-printable characters" "'\aL\CR'"
```

B.4 code/part2/tests/BoaTests/ExpTests.hs

```
module BoaTests.ExpTests where

import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaTests.Util

expTests :: TestTree
expTests = testGroup ">>>> Exp tests"
  [identAndVarTests
  ,operTests
  ,notTests
  ,callTests
  ,listTests
  ,comprTests]


notTests = testGroup ">> Not expression tests"
  [notTest0, notTest1, notTest2, notTest3, notTest4,
   notTest5, notTest6, notTest7, notTest8, notTest9]

identAndVarTests = testGroup ">> Ident, and Var expression tests"
  [identTest0, identTest1, identTest2, identTest3, identTest4, identTest5,
   identTest6, identTest7, identTest8, identTest9, identTest10, identTest11,
   identTest12, identTest13, identTest14, identTest15]

operTests = testGroup ">> Oper expression tests"
  [relationalBinopTests, arithmeticBinopTests]
arithmeticBinopTests = testGroup ">> Arithmetic binop expression tests"
  [plusTest0, plusTest1, minusTest0, minusTest1, timesTest0,
   timesTest1, divTest0, divTest1, modTest0, modTest1]
relationalBinopTests = testGroup ">> Relational binop expression tests"
  [eqTest0, lessTest0, greaterTest0, inTest0,
   neqTest0, lessEqTest0, greaterEqTest0, notInTest0,
   eqTest1, lessTest1, greaterTest1, inTest1,
   neqTest1, lessEqTest1, greaterEqTest1, notInTest1]

callTests = testGroup ">> Call expression tests"
  [callTest0, callTest1, callTest2, callTest3, callTest4,
   callTest5, callTest6, callTest7, callTest8, callTest9]

listTests = testGroup ">> List expression tests"
```

```
  [listTest0, listTest1, listTest2, listTest3, listTest4, listTest5, listTest6]

comprTests = testGroup ">> Comprehension expression tests"
  [comprTest0, comprTest1, comprTest2, comprTest3, comprTest4, comprTest5,
   comprTest6, comprTest7, comprTest8, comprTest9, comprTest10]


identTest0 = test "simple ident" "foo" [SExp (Var "foo")]
identTest1 = test "leading whitespace" " \n foo" [SExp (Var "foo")]
identTest2 = test "trailing whitespace" "foo \t\t\t" [SExp (Var "foo")]
identTest3 = test "leading underscore" " _foo" [SExp (Var "_foo")]
identTest4 = test "non-leading numerics" " _f123123oo" [SExp (Var "_f123123oo")]
identTest5 = test "almost a reserved keyword" "foR = 3" [SDef "foR" (Const (IntVal 3))]
identTest6 = test "test correct whitespace after ident" "foo in [foo]" [SExp (Oper In (Var "foo") (List [Var "foo"]))]
identTest7 = test "legal leading/trailing characters 1" "x+y" [SExp (Oper Plus (Var "x") (Var "y"))]
identTest8 = test "legal leading/trailing characters 2" "x+(y)" [SExp (Oper Plus (Var "x") (Var "y"))]
identTest9 = test "legal leading/trailing characters 3" "x(_y)" [SExp (Call "x" [Var "_y"])]
identTest10 = negTest "leading numerics" "42foo = -42"
identTest11 = negTest "reserved keyword" "for = 3"
identTest12 = negTest "illegal characters 1" "!foo = 3"
identTest13 = negTest "illegal characters 2" "f$oo = 3"
identTest14 = negTest "trailing keyword" "fooin [foo]"
identTest15 = negTest "leading keyword" "[foo] inbar"


a = Const (IntVal 1337)
b = Const (IntVal 42)
c = Const (IntVal 3)
plusTest0 = test "simple add expression" "1337+42"  [SExp (Oper Plus a b)]
plusTest1 = test "correct association of add" "1337+42+3" [SExp (Oper Plus (Oper Plus a b) c)]
minusTest0 = test "simple minus expression" "1337-42" [SExp (Oper Minus a b)]
minusTest1 = test "correct association of sub" "1337-42-3" [SExp (Oper Minus (Oper Minus a b) c)]

timesTest0 = test "simple times expression" "1337*42"  [SExp (Oper Times a b)]
timesTest1 = test "correct association of times" "1337*42*3" [SExp (Oper Times (Oper Times a b) c)]

divTest0 = test "simple div expression" "1337//42"  [SExp (Oper Div a b)]
divTest1 = test "correct association of div" "1337//42//3" [SExp (Oper Div (Oper Div a b) c)]

modTest0 = test "simple mod expression" "1337%42"  [SExp (Oper Mod a b)]
modTest1 = test "correct association of mod" "1337%42%3" [SExp (Oper Mod (Oper Mod a b) c)]

eqTest0      = test "simple == expression" "1337 == 42" [SExp (Oper Eq a b)]
lessTest0    = test "simple < expression" "1337 < 42" [SExp (Oper Less a b)]
greaterTest0 = test "simple > expression" "1337 > 42" [SExp (Oper Greater a b)]
inTest0      = test "simple `in` expression" "1337 in [42]" [SExp (Oper In a (List [b]))]

neqTest0       = test "simple != expression" "1337 != 42" [SExp (Not (Oper Eq a b))]
lessEqTest0    = test "simple <= expression" "1337 <= 42" [SExp (Not (Oper Greater a b))]
greaterEqTest0 = test "simple >= expression" "1337 >= 42" [SExp (Not (Oper Less a b))]
notInTest0      = test "simple `not in` expression" "1337 not in [42]" [SExp (Not (Oper In a (List [b])))]

eqTest1      = negTest "correct non-assocation of ==" "1337 == 42 == 3"
lessTest1    = negTest "correct non-assocation of <" "1337 < 42 < 3"
greaterTest1 = negTest "correct non-assocation of >" "1337 > 42 > 3"
inTest1      = negTest "correct non-assocation of `in`" "3 in [3] in [True]" -- thank god!

neqTest1       = negTest "correct non-assocation of !=" "1337 != 42 != 3"
lessEqTest1    = negTest "correct non-assocation of <=" "1337 <= 42 <= 3"
greaterEqTest1 = negTest "correct non-assocation of >=" "1337 >= 42 >= 3"
notInTest1     = negTest "correct non-assocation of `not in`" "3 not in [3] not in [True]" -- thank god!


operPrecTest0 = test "inter-arithOp precedence 1" "1337 + 3 * 42"
  [SExp (Oper Plus (Const (IntVal 1337)) (Oper Times (Const (IntVal 3)) (Const (IntVal 42))))]
```

```haskell

operPrecTest1 = test "inter-arithOp precedence 2" "1337 + 3 // 42"
  [SExp (Oper Plus (Const (IntVal 1337)) (Oper Div (Const (IntVal 3)) (Const (IntVal 42))))]

operPrecTest2 = test "inter-arithOp precedence 3" "1337 + 3 % 42"
  [SExp (Oper Plus (Const (IntVal 1337)) (Oper Mod (Const (IntVal 3)) (Const (IntVal 42))))]

operPrecTest3 = test "inter-arithOp precedence 4" "1337 - 3 * 42"
  [SExp (Oper Minus (Const (IntVal 1337)) (Oper Times (Const (IntVal 3)) (Const (IntVal 42))))]

operPrecTest4 = test "inter-arithOp precedence 5" "1337 - 3 // 42"
  [SExp (Oper Minus (Const (IntVal 1337)) (Oper Div (Const (IntVal 3)) (Const (IntVal 42))))]

operPrecTest5 = test "inter-arithOp precedence 6" "1337 - 3 % 42"
  [SExp (Oper Minus (Const (IntVal 1337)) (Oper Mod (Const (IntVal 3)) (Const (IntVal 42))))]

operPrecTest6 = test "relOp/arithOp precedence 1" "143 + 5 < 2 % 2"
  [SExp (Oper Less (Oper Plus (Const (IntVal 143)) (Const (IntVal 5))) (Oper Mod (Const (IntVal 2)) (Const (IntVal 2))))]

operPrecTest7 = test "relOp/arithOp precedence 2" "143 + 5 < 2 % 2"
  [SExp (Oper Less (Oper Plus (Const (IntVal 143)) (Const (IntVal 5))) (Oper Mod (Const (IntVal 2)) (Const (IntVal 2))))]

operPrecTests = testGroup "Inter-Oper precedence tests"
  [operPrecTest0, operPrecTest1, operPrecTest2, operPrecTest3,
   operPrecTest4, operPrecTest5, operPrecTest6, operPrecTest7]


notTest0 = test "simple not exp" "not 0" [SExp (Not (Const (IntVal 0)))]
notTest1 = test "nested not exp" "not not not 0" [SExp (Not (Not (Not (Const (IntVal 0)))))]
notTest2 = test "correct precedence 1" "not 3 + 4" [SExp (Not (Oper Plus (Const (IntVal 3)) (Const (IntVal 4))))]
notTest3 = test "correct precedence 2" "not 3 * 4" [SExp (Not (Oper Times (Const (IntVal 3)) (Const (IntVal 4))))]
notTest4 = test "correct precedence 3" "not 3 < 4" [SExp (Not (Oper Less (Const (IntVal 3)) (Const (IntVal 4))))]
notTest5 = test "correct precedence 4" "not foo(x)" [SExp (Not (Call "foo" [Var "x"]))]
notTest6 = test "not with inequality"  "not 3 != 4" [SExp (Not (Not (Oper Eq (Const (IntVal 3)) (Const (IntVal 4)))))]
notTest7 = test "not with list exp"    "not [3, 4, 5] != 4" [SExp (Not (Not (Oper Eq (List
             [Const (IntVal 3),Const (IntVal 4),Const (IntVal 5)]) (Const (IntVal 4)))))]
notTest8 = negTest "illegal association 1" "not 3 + not 4"
notTest9 = negTest "illegal association 2" "3 % not 4"


fooCallExp = [SExp (Call "main" [Const (IntVal 1), Const (StringVal "hej"), Const NoneVal])]
callTest0 = test "simple function call" "main(1, 'hej', None)" fooCallExp
callTest1 = test "one argument" "main(123)" [SExp (Call "main" [Const (IntVal 123)])]
callTest2 = test "no arguments"         "main()" [SExp (Call "main" [])]
callTest3 = test "whitespace in args" "main( 1 , 'hej' , None )" fooCallExp
callTest4 = test "whitespace everywhere" "main ( 1 , 'hej' , None ) " fooCallExp
callTest5 = test "no whitespace" "main(1,'hej',None)" fooCallExp
callTest6 = negTest "missing arg delimeter" "main(1 'hej', None)"
callTest7 = negTest "missing arg after delimeter" "main(1, 'hej', None,)"
callTest8 = negTest "missing parenthesis 1" "main(1, 'hej', None"
callTest9 = negTest "missing parenthesis 2" "main1, 'hej', None)"


fooListProg = [SExp (List [Const (IntVal 1), Const (IntVal 2), Const (IntVal 3)])]
emptyListProg = [SExp (List [])]
listTest0 = test "simple list" "[1, 2, 3]" fooListProg
listTest1 = test "whitespace all over" " [ 1 , 2 , 3 ] " fooListProg
listTest2 = test "no whitespace" "[1,2,3]" fooListProg
listTest3 = test "empty list" "[]" emptyListProg
listTest4 = negTest "missing delimeter" "[1, 2, 3 4]"
listTest5 = negTest "missing closing bracket" "[1, 2, 3, 4"
listTest6 = negTest "missing opening bracket" "1, 2, 3, 4]"


fooComprExp = [SExp (Compr (List [Const (IntVal 1),Const (IntVal 2),
```

```haskell
               Const (IntVal 3)]) [CCFor "x" (List [Var "a",Var "b",Var "c"])])]
comprTest0 = test "simple comprehension expression" "[[1, 2, 3] for x in [a, b, c]]" fooComprExp
comprTest1 = test "extra whitespace" " [ [1, 2, 3] for x in [a, b, c] ] " fooComprExp
comprTest2 = test "minimal whitespace" "[[1, 2, 3]for x in[a, b, c]]" fooComprExp
comprTest3 = test "interleaved if and for clauses" "[x for y in z if 3 < x for z in a if None]"
               [SExp (Compr (Var "x") [CCFor "y" (Var "z"),CCIf (Oper Less (Const (IntVal 3))
                  (Var "x")),CCFor "z" (Var "a"),CCIf (Const NoneVal)])]
comprTest4 = test "`in` relOp not confused with `for` clause" "[x in y for a in c]"
               [SExp (Compr (Oper In (Var "x") (Var "y")) [CCFor "a" (Var "c")])]
comprTest5 = test "comprehension in comprehension" "[1, 4, None, [x for y in z], True]"
               [SExp (List [Const (IntVal 1),Const (IntVal 4),Const NoneVal,Compr (Var "x")
               [CCFor "y" (Var "z")],Const TrueVal])]
comprTest6 = test "comprehension in comprehension" "[[y] for y in [z+3 for z in [yes]]]"
               [SExp (Compr (List [Var "y"]) [CCFor "y" (Compr (Oper Plus (Var "z")
               (Const (IntVal 3))) [CCFor "z" (List [Var "yes"])])])]
comprTest7  = negTest "missing whitespace 1" "[[1, 2, 3]fory in[a, b, c]]"
comprTest8  = negTest "missing whitespace 2" "[[1, 2, 3]for yin[a, b, c]]"
comprTest9 = negTest "missing whitespace 3" "[xfor yin]"
comprTest10 = negTest "no leading for clause" "[if True]"
```

B.5 code/part2/tests/BoaTests/MiscTests.hs

```haskell
module BoaTests.MiscTests where

import Test.Tasty
import Test.Tasty.HUnit

import BoaAST
import BoaTests.Util

commentTests :: TestTree
commentTests = testGroup ">>>> Comment tests"
  [commentTest0, commentTest1, commentTest2, commentTest3, commentTest4]

progTests :: TestTree
progTests = testGroup ">>>> Prog tests"
  [progTest0, progTest1, progTest2,
   progTest3, progTest4, progTest5]

fooStmt0 = SExp (Var "x")
fooStmt1 = SDef "foo" (Const NoneVal)
fooProg  = [fooStmt0, fooStmt1]

progTest0 = test "simple statement program" "foo = None" [fooStmt1]
progTest1 = test "multiple statements" "x; foo = None" fooProg

progTest2 = negTest "empty program" ""
progTest3 = negTest "missing semicolon" "x foo = None"
progTest4 = negTest "trailing semicolon (why is this invalid though)" "x; foo = None;"
progTest5 = negTest "trailing semicolon (why is this invalid though)" "x; foo = None;"

fooProg2 = [SDef "x" (Const (IntVal 5))]
commentTest0 = test "simple comment" "x = 5#hej der\n" fooProg2
commentTest1 = test "leading and trailing comments" "#hej der\nx = 5#hej der\n" fooProg2
commentTest2 = test "multiple comments" "#hej der\n#hej der\n#hej der\nx = 5#hej der\n#hej der\n" fooProg2
commentTest3 = test "trailing comment, no newline after" "x = 5#hej der" fooProg2
commentTest4 = negTest "just a comment" "#just a comment\n"
```