

# 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

$$\begin{aligned} \text{prog} &:= \text{stmtS} \\ \text{stmtS} &:= \text{stmt} \mid \text{stmt} \text{ ";" } \text{stmtS}' \\ \text{stmtS}' &:= \text{stmtS} \leftarrow (\text{? not sure}) \uparrow \\ \text{stmt} &:= \text{ident} \text{ "=" } \text{expr} \mid \text{expr} \\ \text{expr} &:= \text{term} \text{ eopt} \\ \text{eopt} &:= \text{"+" term eopt} \mid \text{"-"} \text{term eopt} \mid \epsilon \\ \text{term} &:= \text{factor}' \text{ topt} \\ \text{topt} &:= \text{"*"} \text{factor}' \text{ topt} \mid \text{"//"} \text{factor}' \text{ topt} \mid \text{"\%"} \text{factor}' \text{ topt} \mid \epsilon \\ \text{factor}' &:= \text{factor} \text{ fopt} \\ \text{fopt} &:= \text{"==" factor fopt} \mid \text{"<"} \text{factor fopt} \mid \dots \mid \epsilon \\ \text{factor} &:= \text{numConst} \mid \text{stringConst} \mid \text{"None"} \mid \text{"True"} \mid \text{"False"} \mid \\ &\quad \text{ident} \mid \text{"not"} \text{ expr} \mid \text{"(" expr ")"} \mid \text{ident} \text{ "(" exprZ ")" } \mid \\ &\quad \text{"[" exprZ "]" } \mid \text{"[" expr forClause clauseZ "]" } \\ \text{forClause} &:= \text{"for"} \text{ ident} \text{ "in"} \text{ expr} \\ \text{ifClause} &:= \text{"if"} \text{ expr} \quad (\text{? not sure}) \\ \text{clauseZ} &:= \text{forClause clauseZ} \mid \text{ifClause clauseZ} \quad \downarrow \\ \text{exprZ} &:= \text{exprS} \mid \epsilon \\ \text{exprS} &:= \text{expr} \mid \text{exprS}' \quad \leftarrow \\ \text{exprS}' &:= \text{expr} \text{ "," } \text{exprS} \quad \leftarrow \\ \left. \begin{array}{l} \text{ident} := \\ \text{numConst} := \\ \text{stringConst} := \end{array} \right\} &\begin{array}{l} \text{see assignment} \\ \text{text} \end{array} \quad 3 \end{aligned}$$

## 7.2 BOA

```

1  -- Skeleton file for Boa Parser.
2
3  module BoaParser (ParseError, parseString) where
4
5  import BoaAST
6  -- add any other other imports you need
7
8  import Text.ParserCombinators.Parsec
9  import Data.Char
10
11 parseString :: String -> Either ParseError Program
12 parseString = parse parseBoa "parse error"
13
14
15 -- REWRITTEN GRAMMER (or at least, the part that was rewritten)
16 -- Expr    := Expr' NegOp Expr | Expr' Op Expr | Expr'
17
18 -- Expr'   := numConst
19 --           | stringConst
20 --           | None
21 --           | True
22 --           | False
23 --           | ident
24 --           | not Expr
25 --           | ( Expr )
26 --           | ident ( Exprz )
27 --           | [ Exprz ]
28 --           | [ Expr ForClause Clausez ]
29
30 -- Op      := '+' | '-' | '*' | '/' | '%' | "==" | "<" | ">" | "in"
31 -- negOp   := "!=" | "<=" | ">=" | "not in"
32
33 parseBoa :: Parser Program
34 parseBoa = do spaces; p <- stmts; eof; return p
35
36 stmts :: Parser [Stmt]
37 stmts = try (do s <- stmt; symbol ";"; ss <- stmts; return (s:ss))
38       <|> do s <- stmt; return [s]
39
40 stmt :: Parser Stmt
41 stmt = try (do i <- ident; symbol "="; e <- expr; return $ SDef i e)
42       <|> do e <- expr; return (SExp e)
43
44 expr :: Parser Exp
45 expr = lexeme $ try (do e <- expr'; o <- negOp; e' <- expr; return $ Not $ Oper o
46                       e e')
47       <|> try (do e <- expr'; o <- op; e' <- expr; return $ Oper o e e')
48       <|> do expr'
49
50 expr' :: Parser Exp
51 expr' = lexeme $ do n <- numConst; return (Const (IntVal n))
52       <|> do symbol "\"'"; s <- stringConst; return (Const (StringVal s))
53       <|> do symbol "None"; return (Const NoneVal)
54       <|> do symbol "True"; return (Const TrueVal)
55       <|> do symbol "False"; return (Const FalseVal)

```

```

55 <|> try (do string "not"; spaces; e <- expr; return (Not e))
56 <|> try (do i <- ident; symbol "("; es <- exprz; symbol ")"; return (Call i
    es))
57 <|> do i <- ident; return (Var i)
58 <|> do symbol "("; e <- expr; symbol ")"; return e
59 <|> try (do symbol "["; es <- exprz; symbol "]"; return (List es))
60 <|> do symbol "["; e <- expr; fc <- forClause; cs <- clausez; symbol "]";
    return (Compr e (fc:cs))
61
62 op :: Parser Op
63 op = lexeme $ do char '+'; return Plus
64 <|> do char '-'; return Minus
65 <|> do char '*'; return Times
66 <|> do string "//"; return Div
67 <|> do char '%'; return Mod
68 <|> do string "==" ; return Eq;
69 <|> do char '<'; return Less
70 <|> do char '>'; return Greater
71 <|> do string "in"; requiredSpaces; return In
72
73 negOp :: Parser Op
74 negOp = lexeme $ do string "!="; return Eq
75 <|> do string "<="; return Greater
76 <|> do string ">="; return Less
77 <|> do string "not in"; return In
78
79 forClause :: Parser CClause
80 forClause = do string "for"; requiredSpaces; i <- ident; string "in";
    requiredSpaces; e <- expr; return (CCFor i e)
81
82 ifClause :: Parser CClause
83 ifClause = do string "if"; requiredSpaces; e <- expr; spaces; return (CCIf e)
84
85 requiredSpaces :: Parser ()
86 requiredSpaces = do lookAhead (satisfy isSpace); spaces; return ()
87
88 clausez :: Parser [CClause]
89 clausez = lexeme $ try (do lookAhead (char ')'); return [])
90 <|> do fc <- forClause; cs <- clausez; return (fc:cs)
91 <|> do ic <- ifClause; cs <- clausez; return (ic:cs)
92
93 exprz :: Parser [Exp]
94 exprz = try (do lookAhead (oneOf ")"]); return []
95 <|> do exprs
96
97 exprs :: Parser [Exp]
98 exprs = try (do e <- expr; symbol ","; es <- exprs; return (e:es))
99 <|> do e <- expr; return [e]
100
101 ident :: Parser String
102 ident = lexeme $ do lookAhead (noneOf "0123456789"); many1 isIdent;
103
104 isIdent :: Parser Char
105 isIdent = (isKeyword >> fail "reserved keyword")
106 <|> satisfy (\c -> isAlphaNum c || c == '_'')
107
108 isKeyword :: Parser ()

```

```

109 isKeyword = try (do string "None"; spaces; oneOf "="; return ())
110     <|> try (do string "True"; spaces; oneOf "="; return ())
111     <|> try (do string "False"; spaces; oneOf "="; return ())
112     <|> try (do string "for"; spaces; oneOf "="; return ())
113     <|> try (do string "if"; spaces; oneOf "="; return ())
114     <|> try (do string "in"; spaces; oneOf "="; return ())
115     <|> try (do string "not"; spaces; oneOf "="; return ())
116
117 numConst :: Parser Int
118 numConst = lexeme $ do string "-"; ds <- getDigits; return (read ("-"+ds))
119     <|> do ds <- getDigits; return (read ds)
120
121 getDigits :: Parser String
122 getDigits = do char '0'; return "0"
123     <|> do lookAhead (noneOf "0"); many1 digit;
124
125 stringConst :: Parser String
126 stringConst = do manyTill isStringConst (char '\\')
127
128 isStringConst :: Parser Char
129 isStringConst = newline
130     <|> (string "\\\\" >> return '\\')
131     <|> (string "\\\'" >> return '\')
132     <|> (char '\\\' >> fail "single backslash")
133     <|> do lookAhead $ satisfy isPrint; satisfy isAscii
134
135 comment :: Parser ()
136 comment = try (do char '#'; skipMany (noneOf "\\"); char '\n'; return ())
137     <|> do char '#'; skipMany (noneOf "\\"); eof
138
139 lexeme :: Parser a -> Parser a
140 lexeme p = try (do a <- p; comment; return a)
141     <|> do a <- p; spaces; return a
142
143 symbol :: String -> Parser ()
144 symbol s = lexeme $ do string s; return ()

```

Listing 1: BoaInterp.hs

## 7.3 Test of BOA

```
1  -- Rudimentary test suite. Feel free to replace anything.
2
3  import BoaAST
4  import BoaParser
5
6  import Test.Tasty
7  import Test.Tasty.HUnit
8
9  main :: IO ()
10 main = defaultMain $ localOption (mkTimeout 1000000) tests
11
12 tests = testGroup "Parser tests" [
13   -- testCase "simple success" $
14   --   parseString "2 + two" @?=
15   --     Right [SExp (Oper Plus (Const (IntVal 2)) (Var "two"))],
16   -- testCase "simple failure" $
17   --   -- avoid "expecting" very specific parse-error messages
18   --   case parseString "wow!" of
19   --     Left e -> return () -- any message is OK
20   --     Right p -> assertFailure $ "Unexpected parse: " ++ show p]
21
22   -- space tests
23   testCase "space at start" $
24     parseString " x" @?=
25       Right [SExp (Var "x")],
26   testCase "tab at start" $
27     parseString "\tx" @?=
28       Right [SExp (Var "x")],
29   testCase "newline at start" $
30     parseString "\nx" @?=
31       Right [SExp (Var "x")],
32
33   -- numConst tests
34   testCase "1" $
35     parseString "1" @?=
36       Right [SExp (Const (IntVal 1))],
37   testCase "-1" $
38     parseString "-1" @?=
39       Right [SExp (Const (IntVal (-1)))],
40   testCase "0" $
41     parseString " 0 " @?=
42       Right [SExp (Const (IntVal (0)))],
43   testCase "-0" $
44     parseString "-0" @?=
45       Right [SExp (Const (IntVal (-0)))],
46   testCase "int overflow" $
47     parseString "18446744073709551615" @?=
48       Right [SExp (Const (IntVal (-1)))],
49   testCase "00" $
50     case parseString "00" of
51       Left e -> return ()
52       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
53   testCase "007" $
54     case parseString "007" of
55       Left e -> return ()
```

```

56     Right p -> assertFailure $ "Unexpected parse: " ++ show p,
57   testCase "- 1" $
58     case parseString "- 1" of
59       Left e -> return ()
60       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
61   testCase "+1" $
62     case parseString "+1" of
63       Left e -> return ()
64       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
65
66   -- ident tests
67   testCase "x" $
68     parseString "x" @?=
69       Right [SExp (Var "x")],
70   testCase "x=1" $
71     parseString "x=1" @?=
72       Right [SDef "x" (Const (IntVal 1))],
73   testCase "x = 1" $
74     parseString "x = 1" @?=
75       Right [SDef "x" (Const (IntVal 1))],
76   testCase "var" $
77     parseString "var" @?=
78       Right [SExp (Var "var")],
79   testCase "not1" $
80     parseString "not1" @?=
81       Right [SExp (Var "not1")],
82   testCase "false1" $
83     parseString "false1" @?=
84       Right [SExp (Var "false1")],
85   testCase "var_1" $
86     parseString "var_1" @?=
87       Right [SExp (Var "var_1")],
88   testCase "_var1" $
89     parseString "_var1" @?=
90       Right [SExp (Var "_var1")],
91   testCase "1_var" $
92     case parseString "1_var" of
93       Left e -> return ()
94       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
95   --reserved words in ident
96   testCase "None=1" $
97     case parseString "None=1" of
98       Left e -> return ()
99       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
100   testCase "True = 1" $
101     case parseString "True = 1" of
102       Left e -> return ()
103       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
104   testCase " False = 1" $
105     case parseString " False = 1" of
106       Left e -> return ()
107       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
108   testCase "for" $
109     case parseString "for" of
110       Left e -> return ()
111       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
112   testCase "if()" $

```



```

113     case parseString "if()" of
114       Left e -> return ()
115       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
116   testCase "in=1" $
117     case parseString "in=1" of
118       Left e -> return ()
119       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
120   testCase "not=1" $
121     case parseString "not=1" of
122       Left e -> return ()
123       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
124
125   -- stringConst tests
126   testCase "'a'" $
127     parseString "\"a\"" @?=
128       Right [SExp (Const (StringVal "a"))],
129   testCase "'hello world'" $
130     parseString "\"hello world\"" @?=
131       Right [SExp (Const (StringVal "hello world"))],
132   testCase "'!\"#$%&()*+,-./:;<=>?[]^_`{|~'" $ --TODO: ascii delete is
133     printable?
134     parseString "'!\"#$%&()*+,-./:;<=>?[]^_`{|~'" @?=
135       Right [SExp (Const (StringVal "!\"#$%&()*+,-./:;<=>?[]^_`{|~"))],
136   testCase "x\\y\\z'" $
137     parseString "x\\\\y\\\\z'" @?=
138       Right [SExp (Const (StringVal "x\\y\\z"))],
139   testCase "x\\y\\'z'" $
140     parseString "x\\\\y\\'z'" @?=
141       Right [SExp (Const (StringVal "x\\y'z"))],
142   testCase "'a\\nb'" $
143     parseString "'a\\\\nb'" @?=
144       Right [SExp (Const (StringVal "a\\nb"))],
145   testCase "'fo\\\\\\\\o\\\\b\\\\na\\\\\\\\r'" $
146     parseString "'fo\\\\\\\\o\\\\b\\\\na\\\\\\\\r'" @?=
147       Right [SExp (Const (StringVal "fo\\\\ob\\na'r"))],
148   testCase "'a\"b\\\\n'" $
149     parseString "'a\\\\\"b\\\\n'" @?=
150       Right [SExp (Const (StringVal "a\"b\\\\n"))],
151   testCase "'\\n'" $
152     parseString "'\\\\n'" @?=
153       Right [SExp (Const (StringVal "\\n"))],
154   testCase "' ' '" $
155     parseString "' ' '" @?=
156       Right [SExp (Const (StringVal "\\235"))],
157   testCase "'\\t'" $
158     case parseString "'\\t'" of
159       Left e -> return ()
160       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
161   testCase "'\\DEL'" $
162     case parseString "'\\DEL'" of
163       Left e -> return ()
164       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
165   testCase "'\\'" $
166     case parseString "'\\'" of
167       Left e -> return ()
168       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
169   testCase "'\\,''" $

```

```

169     case parseString "'\'," of
170       Left e -> return ()
171       Right p -> assertFailure $ "Unexpected parse: " ++ show p,
172
173   -- other expr tests
174   testCase "NoneVal" $
175     parseString "None" @?=
176       Right [SExp (Const NoneVal)],
177   testCase "TrueVal" $
178     parseString "True" @?=
179       Right [SExp (Const TrueVal)],
180   testCase "FalseVal" $
181     parseString "False" @?=
182       Right [SExp (Const FalseVal)],
183   -- oper tested comprehensively below
184   testCase "not 1" $
185     parseString "not 1" @?=
186       Right [SExp (Not (Const (IntVal 1)))],
187   testCase "not(1)" $
188     parseString "not(1)" @?=
189       Right [SExp (Not (Const (IntVal 1)))],
190   testCase "not not 1" $
191     parseString "not not 1" @?=
192       Right [SExp (Not (Not (Const (IntVal 1))))],
193   testCase "(1)" $
194     parseString "(1)" @?=
195       Right [SExp (Const (IntVal 1))],
196   testCase "((1))" $
197     parseString "((1))" @?=
198       Right [SExp (Const (IntVal 1))],
199   testCase "( 1 )" $
200     parseString "( 1 )" @?=
201       Right [SExp (Const (IntVal 1))],
202   testCase "f()" $
203     parseString "f()" @?=
204       Right [SExp (Call "f" [])],
205   testCase "f(x)" $
206     parseString "f(x)" @?=
207       Right [SExp (Call "f" [Var "x"])],
208   testCase "f ( x < 4 )" $
209     parseString "f ( x < 4 )" @?=
210       Right [SExp (Call "f" [Oper Less (Var "x") (Const (IntVal 4))])],
211   testCase "[]" $
212     parseString "[]" @?=
213       Right [SExp (List [])],
214   testCase "[1]" $
215     parseString "[1]" @?=
216       Right [SExp (List [Const (IntVal 1)])],
217   testCase "[1,2]" $
218     parseString "[1,2]" @?=
219       Right [SExp (List [Const (IntVal 1), Const (IntVal 2)])],
220   testCase "[1, 2, 3]" $
221     parseString "[1,2,3]" @?=
222     Right [SExp (List [Const (IntVal 1), Const (IntVal 2), Const (IntVal 3)])]
223 ],
224   testCase "[(1+2), True, 'yes']" $
225     parseString "[(1+2), True, 'yes']" @?=

```

```

225     Right [SExp (List [Oper Plus (Const (IntVal 1)) (Const (IntVal 2)),
Const (TrueVal), Const (StringVal "yes")]]],
226     testCase "[x!=y,x>=y,x<=y,x not in y]" $
227       parseString "[x!=y,x>=y,x<=y,x not in y]" @?=
228       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"))]]],
229
230   -- oper tests
231   testCase "1+1" $
232     parseString "1+1" @?=
233     Right [SExp (Oper Plus (Const (IntVal 1)) (Const (IntVal 1)))],
234   testCase "1-1" $
235     parseString "1-1" @?=
236     Right [SExp (Oper Minus (Const (IntVal 1)) (Const (IntVal 1)))],
237   testCase "1 * 1" $
238     parseString "1 * 1" @?=
239     Right [SExp (Oper Times (Const (IntVal 1)) (Const (IntVal 1)))],
240   testCase "1 // 1" $
241     parseString "1 // 1" @?=
242     Right [SExp (Oper Div (Const (IntVal 1)) (Const (IntVal 1)))],
243   testCase "1 % 1" $
244     parseString "1 % 1" @?=
245     Right [SExp (Oper Mod (Const (IntVal 1)) (Const (IntVal 1)))],
246   testCase "1 == True" $
247     parseString "1 == True" @?=
248     Right [SExp (Oper Eq (Const (IntVal 1)) (Const TrueVal))],
249   testCase "1 != 1" $
250     parseString "1 != 1" @?=
251     Right [SExp (Not (Oper Eq (Const (IntVal 1)) (Const (IntVal 1))))],
252   testCase "1 < 1" $
253     parseString "1 < 1" @?=
254     Right [SExp (Oper Less (Const (IntVal 1)) (Const (IntVal 1)))],
255   testCase "1 <= 1" $
256     parseString "1 <= 1" @?=
257     Right [SExp (Not (Oper Greater (Const (IntVal 1)) (Const (IntVal 1))))]
],
258   testCase "1 > 1" $
259     parseString "1 > 1" @?=
260     Right [SExp (Oper Greater (Const (IntVal 1)) (Const (IntVal 1)))],
261   testCase "1 >= 1" $
262     parseString "1 >= 1" @?=
263     Right [SExp (Not (Oper Less (Const (IntVal 1)) (Const (IntVal 1))))],
264   testCase "1 in [1]" $
265     parseString "1 in [1]" @?=
266     Right [SExp (Oper In (Const (IntVal 1)) (List [Const (IntVal 1)]))],
267   testCase "1 not in [1]" $
268     parseString "1 not in [1]" @?=
269     Right [SExp (Not (Oper In (Const (IntVal 1)) (List [Const (IntVal 1)]))
)],
270
271   -- clausez tests
272   testCase "[ 1for x in x ]" $
273     parseString "[ 1for x in x ]" @?=
274     Right [SExp (Compr (Const (IntVal 1)) [CCFor "x" (Var "x")])],
275   testCase "[ 1 for x in x if 1 == 1]" $
276     parseString "[ 1 for x in x if t == 1]" @?=

```

```

277     Right [SExp (Compr (Const (IntVal 1)) [CCFor "x" (Var "x"), CCIf (Oper
Eq (Var "t") (Const (IntVal 1)))]),
278     testCase "[if x]" $
279         case parseString "[if x]" of
280             Left e -> return ()
281             Right p -> assertFailure $ "Unexpected parse: " ++ show p,
282
283     -- keyword space tests
284     testCase "2 in[1]" $
285         case parseString "2 in[1]" of
286             Left e -> return ()
287             Right p -> assertFailure $ "Unexpected parse: " ++ show p,
288     testCase "[1 forx in x]" $
289         case parseString "[1 forx in x]" of
290             Left e -> return ()
291             Right p -> assertFailure $ "Unexpected parse: " ++ show p,
292     testCase "[1 for x inx]" $
293         case parseString "[1 for x inx]" of
294             Left e -> return ()
295             Right p -> assertFailure $ "Unexpected parse: " ++ show p,
296     testCase "[1for x in x ifx]" $
297         case parseString "[1for x in x ifx]" of
298             Left e -> return ()
299             Right p -> assertFailure $ "Unexpected parse: " ++ show p,
300
301     -- associativity tests
302     testCase "1-2+3" $
303         parseString "1-2+3" @?=
304         Right [SExp (Oper Plus (Oper Minus (Const (IntVal 1)) (Const (IntVal 2))
) (Const (IntVal 3)))],
305     testCase "(1-2)+3" $
306         parseString "(1-2)+3" @?=
307         Right [SExp (Oper Plus (Oper Minus (Const (IntVal 1)) (Const (IntVal 2))
) (Const (IntVal 3)))],
308     testCase "1-(2+3)" $
309         parseString "1-(2+3)" @?=
310         Right [SExp (Oper Minus (Const (IntVal 1)) (Oper Plus (Const (IntVal 2))
) (Const (IntVal 3)))],
311     testCase "1*2//3" $
312         parseString "1*2//3" @?=
313         Right [SExp (Oper Div (Oper Times (Const (IntVal 1)) (Const (IntVal 2))
) (Const (IntVal 3)))],
314     testCase "(1*2)//3" $
315         parseString "(1*2)//3" @?=
316         Right [SExp (Oper Div (Oper Times (Const (IntVal 1)) (Const (IntVal 2))
) (Const (IntVal 3)))],
317     testCase "1*(2//3)" $
318         parseString "1*(2//3)" @?=
319         Right [SExp (Oper Times (Const (IntVal 1)) (Oper Div (Const (IntVal 2))
(Const (IntVal 3)))],
320     testCase "x<y<z" $
321         case parseString "x<y<z" of
322             Left e -> return ()
323             Right p -> assertFailure $ "Unexpected parse: " ++ show p,
324
325     -- statement tests
326     testCase "x; x=1;1" $

```

```

327     parseString "x; x=1;1" @?=
328         Right [SExp (Var "x"), SDef "x" (Const (IntVal 1)), SExp (Const (IntVal
329         1))],
330
331     -- comment tests
332     testCase "x#bar" $
333         parseString "x#bar" @?=
334             Right [SExp (Var "x")],
335     testCase "x#bar\\n" $
336         parseString "x#bar\\n" @?=
337             Right [SExp (Var "x")],
338     testCase "x#\\n" $
339         parseString "x#\\n" @?=
340             Right [SExp (Var "x")],
341     testCase "x#\\n=1" $
342         parseString "x#\\nbar" @?=
343             Right [SDef "x" (Const (IntVal 1))],
344     testCase "x#bar 78 \\n = 1" $
345         parseString "x#bar 78 \\n = 1" @?=
346             Right [SDef "x" (Const (IntVal 1))]

```

Listing 2: Test.hs