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 := stats
  stmt5 := stmt | stmt ";" stmts'
 stmts' = stmts < (3 not sure)
 stmt := ident "=" expr | expr
 expr := term eopt
 egpt := "+" term copt | "-" term copt | &
term := factor' topt
 topt := "*" factor topt | "/" factor topt | "% factor topt | E
 factor := factor fopt
fopt := "==" factor fopt | "Z" factor fopt | .... | E
 factor := num (onst | string(onst | "None" | "True" | "False" |
             ident | "not" expr | " (" expr")" | ident " ("expr?")"
             "[" exprz "]" ["expr for (lause clause? "]"
for (lause := "for" ident "in" expr
clause 2 := for (lause clause 2) if (lause dause?)

expr2 := exprS | F
exprS := expr /exprS' K
exprs' := expr "," exprs
ident := 7

num(onst := 5 see assignment 3

string (onst := 7)
```

7.2 BOA

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

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

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

Listing 1: BoaInterp.hs

7.3 Test of BOA

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

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

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

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

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

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

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

Listing 2: Test.hs