Haskell intro

Assignment 2

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 26, 2018



Contents

1	Design/Implementation		
	1.1	Choice of Parser combinator Library	1
	1.2	Whitespace	1
	1.3	Precedence and Associativity	2
2	Code Assessment		
	2.1	Tests	2
	2.2	Test Coverage	2
A	A Code Listing		3
В	Test	S	9

1 Design/Implementation

1.1 Choice of Parser combinator Library

We decided to use **Parsec** because of the better error handling capabailities compared to **ReadP**.

1.2 Whitespace

We decided to remove leading whitespace and trailing whitespace to parse the tokens, this we do by using a function 'parseLeadingWhitespace' and 'parse-Whitespace' which takes care of removing the whitespaces, newlines and other characters in strings as well.

1.3 Precedence and Associativity

2 Code Assessment

We are relatively confident that we were able to program a more or less working parser for Subscript, also thanks to our Tests which should test most cases or at least those we came up with. Nonetheless the Code seems to get less readable since everything is grouped in one single file, same for the tests which end up to be quite long (testing on string for ParseErrors also doen't seem like a nice solution, but we didn't came up with a better one). Our way of cope with the overall complexity was by trying grouping the function which belong together as good as possible but there definitely would be a nicer solution available.

2.1 Tests

We wrote overall 70 Tests which either Test more Complex expressions or the very basic functionality of the parser. For that we also had to write a 'ParserUtils.hs' file which has some utilities for calling the actual functions for testing (like 'parseNumber'), for ArrayComprehensions on the other side we used the 'parseString' function directly since we walked into the Problem that calling 'parseArrayCompr' wasn't possible right away, so we went the easy way and used 'parseString'.

2.2 Test Coverage

Our test coverage is quite high and pretty much should test all cases possible, at least 97% of epxpressions are used:

- 97% expressions used (448/458)
- 63% boolean coverage (7/11)
- 50% guards (4/8), 2 always True, 1 always False, 1 unevaluated
- 100% 'if' conditions (3/3)
- 100% qualifiers (0/0)
- 83% alternatives used (15/18)
- 100% local declarations used (1/1)
- 88% top-level declarations used (46/52)

A Code Listing

```
module Parser.Impl where
   -- Put your parser implementation in this file (and, if
    → appropriate,
  -- in other files in the Parser/ subdirectory)
   import SubsAst
   import Text.Parsec.Char
   import Text.Parsec.Combinator
   import Text.Parsec.Prim
  import Text.Parsec.String
10
  -- ord used in isLegalChar to check if char is in printable ASCII
11
    → range
  import Data.Char
12
13
   data ParseError =
     ParseError String
     deriving (Eq, Show)
16
17
   parseString :: String -> Either ParseError Expr
   parseString s =
19
     case parse
20
             (do res <- parseLeadingWhitespace parseExpr</pre>
21
                 eof
22
                 return res)
23
             "ERROR"
             s of
25
       Left a -> Left (ParseError (show a))
26
       Right b -> Right b
27
28
  posNumber :: Parser Expr
29
   posNumber = do
30
     n <- many1 digit
31
     if length n <= 8</pre>
32
       then return $ Number $ read n
33
       else fail "Number too long"
34
35
   negNumber :: Parser Expr
36
   negNumber = do
37
     m <- string "-"
38
     n <- many1 digit
39
     if length n <= 8</pre>
```

```
then return $ Number $ read (m ++ n)
41
       else fail "Number too long"
43
  parseNumber :: Parser Expr
44
45
   parseNumber = do
     parseWhitespace (posNumber <|> negNumber)
46
47
   parseParentheses :: Parser Expr
48
   parseParentheses = do
49
     _ <- parseWhitespace (char '(')</pre>
50
    expr <- parseExpr
51
     _ <- parseWhitespace (char ')')</pre>
52
     return expr
53
54
55
  parseComment :: Parser ()
  parseComment = do
56
    _ <- string "//"
57
     _ <- manyTill anyChar (newLine <|> eof)
     return ()
59
60
   --makes newline be of type ()
61
62 newLine :: Parser ()
   newLine = do
63
     _ <- newline
64
     return ()
65
66
  parseLeadingWhitespace :: Parser a -> Parser a
67
   parseLeadingWhitespace par = do
     spaces
69
     optional parseComment
70
71
     spaces
     par
72
73
  parseWhitespace :: Parser a -> Parser a
74
   parseWhitespace par = do
75
     p <- par
76
77
     spaces
     optional parseComment
78
     spaces
79
     return p
80
81
82 -- check for comma
   parseExpr :: Parser Expr
   parseExpr = choice [parseNotComma, parseCons]
```

```
85
   parseNotComma :: Parser Expr
86
   parseNotComma = do
87
      expr1 <- parseWhitespace parseExpr'</pre>
88
89
      parseComma expr1
90
   parseComma :: Expr -> Parser Expr
91
   parseComma expr1 =
92
      (do _ <- parseWhitespace (char ',')</pre>
93
          expr2 <- parseWhitespace parseExpr</pre>
94
95
          return (Comma expr1 expr2)) <|>
      return expr1
96
97
   keywords :: [String]
98
   keywords = ["true", "false", "undefined", "for", "of", "if"]
100
   parseCons :: Parser Expr
101
   parseCons =
102
      choice
103
        [ try parseArray
104
        , parseArrayStart
105
        , try parseCall
106
        , parseParentheses
107
        , parseNumber
108
        , parseStr
109
        , parseTrue
110
        , parseFalse
111
        , parseUndefined
112
        , try parseAssign
113
        , try parseIdent
114
115
116
   parseIdent :: Parser Expr
117
   parseIdent = do
     fc <- letter
119
      rest <- many (digit <|> letter <|> char '_')
120
      let input = fc : rest
      if input 'notElem' keywords
122
        then return (Var input)
123
        else fail "should not be a keyword"
124
125
   parseAssign :: Parser Expr
126
   parseAssign = do
127
      Var ident <- parseWhitespace parseIdent</pre>
```

```
_ <- parseWhitespace (char '=')</pre>
129
      expr1 <- parseExpr'</pre>
130
      return (Assign ident expr1)
131
132
    parseCall :: Parser Expr
133
    parseCall = do
      Var ident <- parseWhitespace parseIdent</pre>
135
      _ <- parseWhitespace (char '(')</pre>
136
      exprs <- parseExprs</pre>
137
      _ <- parseWhitespace (char ')')</pre>
138
      return (Call ident exprs)
139
140
    parseExprs :: Parser [Expr]
141
    parseExprs =
142
      do expr1 <- parseExpr'</pre>
143
         parseCommaExprs expr1
144
          <|> return []
145
146
    parseCommaExprs :: Expr -> Parser [Expr]
147
    parseCommaExprs expr1 =
148
      do _ <- parseWhitespace (char ',')</pre>
149
         expr2 <- parseExprs</pre>
150
         return (expr1 : expr2)
151
          <|> return [expr1]
152
153
    parseArrayStart :: Parser Expr
154
    parseArrayStart = do
155
      _ <- parseWhitespace (char '[')</pre>
      compr <- parseArrayFor</pre>
157
      _ <- parseWhitespace (char ']')</pre>
158
      return (Compr compr)
159
160
    parseArrayFor :: Parser ArrayCompr
161
162
    parseArrayFor = do
      _ <- parseWhitespace (string "for")</pre>
163
      _ <- parseWhitespace (char '(')
164
      Var ident <- parseWhitespace parseIdent</pre>
165
      _ <- parseWhitespace (string "of")</pre>
      expr1 <- parseWhitespace parseExpr'</pre>
167
      _ <- parseWhitespace (char ')')</pre>
168
      compr <- parseArrayCompr</pre>
169
      return (ACFor ident exprl compr)
170
171
   parseArrayCompr :: Parser ArrayCompr
```

```
parseArrayCompr = choice [try parseACBody, parseArrayFor,

→ parseACIf]
174
   parseACBody :: Parser ArrayCompr
175
176
   parseACBody = do
      expr <- parseExpr'
177
      return (ACBody expr)
178
179
   parseACIf :: Parser ArrayCompr
180
   parseACIf = do
181
      _ <- parseWhitespace (string "if")</pre>
182
      _ <- parseWhitespace (char '(')</pre>
183
      expr1 <- parseWhitespace parseExpr'</pre>
184
      _ <- parseWhitespace (char ')')</pre>
185
      compr <- parseArrayCompr</pre>
      return (ACIf exprl compr)
187
188
   parseArray :: Parser Expr
   parseArray = do
190
      _ <- parseWhitespace (char '[')</pre>
191
      exprs <- parseExprs</pre>
192
      _ <- parseWhitespace (char ']')</pre>
193
     return (Array exprs)
194
   -- checks that the char after the backslash is one of the legal
     → possibilites
   isLegalAfterBackslash :: Char -> Either ParseError Char
197
   isLegalAfterBackslash c
     | c == 'n' = Right '\n'
199
     | c == 't' = Right '\t'
200
      | c 'elem' ['\'', '\\'] = Right c
      | otherwise = fail "Backslash followed by invalid char"
202
203
    -- extracts the char after the \ to return it together with \
   isLegalBackslash :: Parser Char
205
   isLegalBackslash = do
206
      _ <- char '\\'
      c <- oneOf ['\'', 'n', 't', '\\']</pre>
208
      case isLegalAfterBackslash c of
209
        Right a -> return a
210
        _ -> fail "Fail in Backslash"
211
212
    -- checks for printable ascii chars and not \' and \\
213
   isLegalChar :: Char -> Bool
```

```
isLegalChar c
215
     | c == '\'' = False
216
      | c == '\\' = False
217
     | ord c >= 32 && ord c <= 126 = True
218
219
     | otherwise = False
220
    -- option""(try) checks for newline in string to be skipped
221
    -- then checks for backslashes and legal chars
222
   parseCharInStr :: Parser Char
   parseCharInStr = do
224
     _ <- option "" (try (string "\\\n"))</pre>
225
     a <- isLegalBackslash <|> satisfy isLegalChar
      _ <- option "" (try (string "\\\n"))</pre>
227
     return a
228
229
230 parseStr :: Parser Expr
   parseStr = do
231
     _ <- char '\''
     res <- many parseCharInStr
233
      _ <- parseWhitespace (char '\'')</pre>
234
     return (SubsAst.String res)
235
236
237 parseTrue :: Parser Expr
238 parseTrue = do
     _ <- string "true"</pre>
239
     return TrueConst
240
241
242 parseFalse :: Parser Expr
   parseFalse = do
243
      _ <- string "false"</pre>
244
     return FalseConst
245
246
247 parseUndefined :: Parser Expr
   parseUndefined = do
     _ <- string "undefined"</pre>
249
     return Undefined
250
252 parseExpr' :: Parser Expr
   parseExpr' = parseAdditon 'chainl1' parseCompare
253
254
   parseCompare :: Parser (Expr -> Expr -> Expr)
255
   parseCompare =
256
      (do _ <- parseWhitespace (string "<")</pre>
257
          return (\x y -> Call "<" [x, y])) <|>
```

```
(do _ <- parseWhitespace (string "===")</pre>
259
           return (\langle x y \rightarrow Call "===" [x, y] \rangle)
260
261
   parseAdditon :: Parser Expr
262
263
   parseAdditon = do
      prod <- parseWhitespace parseProd</pre>
      parseAdditon' prod
265
266
   parseAdditon' :: Expr -> Parser Expr
   parseAdditon' input =
268
      (do addOp <- parseWhitespace (char '+' <|> char '-')
269
          cons <- parseProd</pre>
270
          parseAdditon' $ Call [addOp] [input, cons]) <|>
271
      return input
272
273
   parseProd :: Parser Expr
274
   parseProd = do
275
      cons <- parseWhitespace parseCons</pre>
      parseProd' cons
277
278
   parseProd' :: Expr -> Parser Expr
   parseProd' input =
280
      (do prodOp <- parseWhitespace (char '*' <|> char '%')
281
          cons <- parseCons</pre>
282
          parseProd' $ Call [prodOp] [input, cons]) <|>
283
      return input
284
```

B Tests

```
testGroup
13
       "Tests"
       [ constantTests
15
       , parseNumberTests
16
17
       , parseStringTests
       , parseFalseTests
18
        , parseTrueTests
19
       , parseUndefinedTests
20
       , parseAssignTests
21
       , parseCallTests
22
       , parseIdentTests
23
       , parseArrayTests
24
       , parseStartArrayTests
25
       , parseParanthesTests
26
       , parseExprs
27
       , parseComma
28
       , parseExprTests
29
       , parseArrayCompr
30
       , parseSimpleExprTests
31
       , parseComplexExprTests
32
       , predefinedTests
33
       , parseErrorTest
34
35
36
   parseNumberTests :: TestTree
37
   parseNumberTests =
38
     testGroup
39
       "parse number"
       [ testCase "Number pos" $ numberParser ("1") @?= Right
41
     \hookrightarrow (Number 1)
       , testCase "Number neg" $ numberParser ("-2") @?= Right
42
       (Number (-2))
       , testCase "Number trailing whitespace" $
43
                               ") @?= Right (Number 1)
         numberParser ("1
       , testCase "Number 8 long pos" $
45
         numberParser ("12345678") @?= Right (Number 12345678)
46
       , testCase "Number 8 long neg" $
         numberParser ("-12345678") @?= Right (Number (-12345678))
48
       , testCase "Number too long pos" $
49
         show (numberParser ("123456789")) @?=
50
          "Left \"ERROR\" (line 1, column 10):\nunexpected end of
51
     → input\nexpecting digit\nNumber too long"
       , testCase "Number too long neg" $
52
         show (numberParser ("-123456789")) @?=
53
```

```
"Left \"ERROR\" (line 1, column 11):\nunexpected end of
    → input\nexpecting digit\nNumber too long"
55
       1
56
57
   parseStringTests :: TestTree
   parseStringTests =
58
     testGroup
59
       "parse string"
60
       [ testCase "String" $ stringParser ("'abc'") @?= Right
61
    , testCase "String alphaNum" $
62
         stringParser ("'abc123'") @?= Right (String "abc123")
63
       , testCase "String allowed special chars" $
64
         stringParser ("'abc\\n\\t'") @?= Right (String "abc\n\t")
65
       , testCase "String not allowed special char" $
66
         show (stringParser ("'\\a'")) @?=
67
         "Left \"ERROR\" (line 1, column 3):\nunexpected \"\\a\""
68
       , testCase "String whitespaced" $
69
         stringParser ("'asdas asdasd'") @?= Right (String "asdas
    → asdasd")
       , testCase "String newline" $
71
         stringParser ("'foo\\\nbar'") @?= Right (String "foobar")
72
       , testCase "Not Allowed ASCII character" $
73
         show (stringParser ("''")) @?=
74
         "Left \"ERROR\" (line 1, column 2):\nunexpected
    \rightarrow \"\\252\"\nexpecting \"\\\\\n\", \"\\\\\" or \"'\""
       , testCase "backslash chars" $
76
         stringParser ("'\\t\\n\\'\\\") @?= Right (String
77
    , testCase "string comment" $
78
         stringParser ("'//Comment 123'") @?= Right (String
    \rightarrow "//Comment 123")
       1
80
81
   parseFalseTests :: TestTree
82
   parseFalseTests =
83
     testGroup
85
       "parse false"
       [ testCase "False" $ falseParser ("false") @?= Right
86
    \hookrightarrow (FalseConst)
       , testCase "False fail" $
87
         show (falseParser ("true")) @?=
88
         "Left \"ERROR\" (line 1, column 1):\nunexpected
89
    → \"t\"\nexpecting \"false\""
```

```
90
       ]
   parseTrueTests :: TestTree
92
   parseTrueTests =
93
94
     testGroup
        "parse true"
95
        [ testCase "True" $ trueParser ("true") @?= Right (TrueConst)
96
        , testCase "True fail" $
97
          show (trueParser ("false")) @?=
98
          "Left \"ERROR\" (line 1, column 1): \nunexpected
99
     → \"f\"\nexpecting \"true\""
100
101
   parseUndefinedTests :: TestTree
102
   parseUndefinedTests =
103
     testGroup
104
        "Undefined"
105
        [ testCase "Undefined" $ undefinedParser ("undefined") @?=
106
     → Right (Undefined)
        , testCase "Undefined fail" $
107
          show (undefinedParser ("defined")) @?=
108
          "Left \"ERROR\" (line 1, column 1):\nunexpected
109
     → \"d\"\nexpecting \"undefined\""
110
111
   parseAssignTests :: TestTree
112
   parseAssignTests =
113
114
     testGroup
        "Assign"
115
        [ testCase "Assign" $ assignParser ("x=3") @?= Right (Assign
116
     , testCase "Assign whitespace/special char" $
117
         assignParser ("x = \n 3") @?= Right (Assign "x" (Number 3))
118
        , testCase "Assign underline" $
119
          assignParser ("x_x=0") @?= Right (Assign "x_x" (Number 0))
120
121
123
   parseCallTests :: TestTree
   parseCallTests =
124
     testGroup
125
        "Call"
126
        [ testCase "Call" $ callParser ("x(12)") @?= Right (Call "x"
127
       [Number 12])
        , testCase "Call whitespace" $
```

```
callParser ("x ( 12 ) ") @?= Right (Call "x" [Number 12])
129
130
131
   parseIdentTests :: TestTree
132
133
   parseIdentTests =
     testGroup
134
        "Ident"
135
        [ testCase "Ident" $ identParser ("x_x") @?= Right (Var
136
     \hookrightarrow "X_X")
        , testCase "Ident similar to keyword" $
137
          identParser ("falsee") @?= Right (Var "falsee")
138
        , testCase "Ident keyword" $
139
          show (identParser ("false")) @?=
140
          "Left \"ERROR\" (line 1, column 6):\nunexpected end of
141
     → input\nexpecting digit, letter or \"_\"\nshould not be a

→ keyword"

        , testCase "Ident whitespace" $
142
          show (identParser ("x_x ")) @?=
143
          "Left \"ERROR\" (line 1, column 4):\nunexpected '
144

→ '\nexpecting digit, letter, \"_\" or end of input"

145
146
   parseArrayTests :: TestTree
147
   parseArrayTests =
148
      testGroup
149
        "Array"
150
        [ testCase "Array" $
151
          parseString ("[1,2]") @?= Right (Array [Number 1, Number
152

→ 2])
        , testCase "Array whitespace" $
153
          parseString ("[ 1, 'sds'] ") @?= Right (Array [Number 1,
     1
155
156
   parseStartArrayTests :: TestTree
157
   parseStartArrayTests =
158
     testGroup
159
160
        "Array Compr"
        [ testCase "Array for" $
161
          parseString ("[for (x \text{ of } 2) \text{ 2}]") @?=
162
          Right (Compr (ACFor "x" (Number 2) (ACBody (Number 2)))),
         testCase "Empty Array" $ parseString("[]") @?= Right(Array
164
     ]
```

```
166
   parseParanthesTests :: TestTree
167
    parseParanthesTests =
168
      testGroup
169
        "Parantheses"
170
        [ testCase "Parantheses" $ parseString ("(1)") @?= Right
171
     \hookrightarrow (Number 1)
        , testCase "Parantheses whitespace" $
172
          parseString ("( 1 )") @?= Right (Number 1)
173
        ]
174
175
   parseExprs :: TestTree
176
    parseExprs =
177
      testGroup
178
        "parseExprs"
179
        [ testCase "parseExprs numbers" $
180
          parseString ("[1,2,3]") @?= Right (Array [Number 1, Number
181

→ 2, Number 3])

        , testCase "parseExprs" $
182
          parseString ("['a','b','c']") @?=
183
          Right (Array [String "a", String "b", String "c"])
184
        , testCase "parseExprs ident" $
185
          parseString ("a (1,2,3)") @?=
186
          Right (Call "a" [Number 1, Number 2, Number 3])
187
        ]
188
189
   parseComma :: TestTree
190
191
    parseComma =
      testGroup
192
        "Comma"
193
        [ testCase "Parse Comma" $
          parseString ("1,2") @=? Right (Comma (Number 1) (Number 2))
195
        , testCase "Parse nested commas" $
196
          parseString ("1, (1, (3, 4))") @?=
197
          Right (Comma (Number 1) (Comma (Number 1) (Comma (Number 3)
198
     , testCase "many commas" $
199
          parseString ("1,2,3,'a','b'") @?=
200
          Right
201
             (Comma
202
                (Number 1)
203
                (Comma
204
205
                   (Comma (Number 3) (Comma (String "a") (String
       "b")))))
```

```
207
        ]
208
   parseExprTests :: TestTree
209
   parseExprTests =
210
      testGroup
211
        "parseExpr"
212
        [ testCase "Additon" $
213
          parseString ("1+1") @=? Right (Call "+" [Number 1, Number
214
     → 1])
        , testCase "Subtraction" $
215
          parseString ("1-1") @?= Right (Call "-" [Number 1, Number
216
     → 1])
        , testCase "Mul" $
217
          parseString ("1*1") @?= Right (Call "*" [Number 1, Number
218
     , testCase "Mod" $
219
          parseString ("1%1") @?= Right (Call "%" [Number 1, Number
220
     , testCase "Smaller Then" \$
221
          parseString ("1<1") @?= Right (Call "<" [Number 1, Number</pre>
222
     \rightarrow 11)
        , testCase "Equals" $
223
          parseString ("1===1") @?= Right (Call "===" [Number 1,
224
     → Number 1])
        ]
225
226
   parseArrayCompr :: TestTree
227
228
    parseArrayCompr =
      testGroup
229
        "Array Compr"
230
        [ testCase "for" $
          parseString ("[for (x \text{ of } 2) 3]") @=?
232
          Right (Compr (ACFor "x" (Number 2) (ACBody (Number 3))))
233
        , testCase "nested for" $
234
          parseString ("[for (x \text{ of } 2) \text{ for } (x \text{ of } 3) \text{ 3}]") @=?
235
236
             (Compr (ACFor "x" (Number 2) (ACFor "x" (Number 3)
237
     → (ACBody (Number 3))))
        , testCase "nested if" $
238
          parseString ("[for (x of 2) if(1) 2]") @=?
239
          Right (Compr (ACFor "x" (Number 2) (ACIf (Number 1) (ACBody
     → (Number 2)))))
        , testCase "mixed for/if" $
241
          parseString ("[for (x of 2) if(1) for (y of 2) if(false)
     \rightarrow for(z of 5) 2]") @=?
```

```
Right
243
             (Compr
244
                (ACFor
245
                   ^{\prime\prime} \times ^{\prime\prime}
246
                   (Number 2)
247
                   (ACIf
248
                      (Number 1)
249
                      (ACFor
250
                         "y"
251
                          (Number 2)
252
                          (ACIf FalseConst (ACFor "z" (Number 5)
253
        (ACBody (Number 2)))))))
        1
254
255
   constantTests :: TestTree
256
   constantTests =
257
     testGroup
258
        "constants tests"
        [ testCase "Number" $ parseString ("2") @?= Right (Number 2)
260
        , testCase "String" $ parseString ("'abc'") @?= Right (String
261
     → "abc")
       , testCase "true" $ parseString ("true") @?= Right
262
        (TrueConst)
        , testCase "false" $ parseString ("false") @?= Right
263
     \hookrightarrow (FalseConst)
        , testCase "Undefined" $ parseString ("undefined") @?= Right
264
        (Undefined)
        , testCase "Ident" $ parseString ("sdsd") @?= Right (Var
265

    "sdsd")

        ]
266
   parseSimpleExprTests :: TestTree
268
   parseSimpleExprTests =
269
270
      testGroup
        "Simple expr tests"
271
        [ testCase "equal" $
272
          parseString ("a===b===c") @?=
273
          Right (Call "===" [Call "===" [Var "a", Var "b"], Var "c"])
274
        , testCase "assign" $
275
          parseString ("a=b=undefined") @?=
276
          Right (Assign "a" (Assign "b" Undefined))
277
        , testCase "smallerThen" $
278
          parseString ("2<3<4") @?=</pre>
279
```

```
, testCase "whitespace" $
281
          parseString ("12
                                            \n") @?= Right (Number 12)
                               \v \t\t
282
        , testCase "comment" $
283
          parseString ("1 //comment 11212121212\n,2") @?=
284
          Right (Comma (Number 1) (Number 2))
285
        , testCase "comment at start" $
286
          parseString ("//comment \n 2
                                             ") @?= Right (Number 2)
287
288
289
   parseComplexExprTests :: TestTree
290
    parseComplexExprTests =
291
      testGroup
292
        "Complex expr tests"
293
        [ testCase "scope.js" $
294
          parseString ("x = 42, y = [for (x of 'abc') x], [x, y]") @?=
295
          Right
296
             (Comma
297
                (Assign "x" (Number 42))
298
                (Comma
299
                    (Assign "y" (Compr (ACFor "x" (String "abc")
300
         (ACBody (Var "x")))))
                   (Array [Var "x", Var "y"])))
301
        , testCase "correct precedence add" $
302
          parseString ("[1,2,3,4] + [1,2,3]") @?=
303
          Right
304
             (Call
305
                \Pi \perp \Pi
306
                [ Array [Number 1, Number 2, Number 3, Number 4]
307
                , Array [Number 1, Number 2, Number 3]
308
                ])
309
        , testCase "precedences" $
310
          parseString ("1+2*4-3%4") @?=
311
          Right
312
             (Call
313
314
                [ Call "+" [Number 1, Call "*" [Number 2, Number 4]]
315
                , Call "%" [Number 3, Number 4]
316
317
                ])
        , testCase "arrayCompr complex" $
318
          parseString
319
             ("[for (a of 4) 1] * [for (a of abc) if (true) if (false)
320
       2*3]") @=?
          Right
321
322
             (Call
```

```
11 <sub>*</sub> 11
323
               [ Compr (ACFor "a" (Number 4) (ACBody (Number 1)))
324
               , Compr
325
                   (ACFor
326
                      "a"
327
                       (Var "abc")
328
                       (ACIf
329
                         TrueConst
330
                          (ACIf FalseConst (ACBody (Call "*" [Number
331
     → 2, Number 3])))))
332
               1)
        ]
333
334
   parseErrorTest :: TestTree
335
   parseErrorTest =
336
     testGroup
337
        "Parse Fail"
338
        [ testCase "let parser fail" $
339
          show (parseString ("")) @=?
340
          "Left (ParseError \"\\\"ERROR\\\" (line 1, column
341
       1):\\nunexpected end of input\\nexpecting white space,
     \\\"'\\\", \\\"true\\\", \\\"false\\\" or
     → \\\"undefined\\\"\")"
       1
342
343
   predefinedTests :: TestTree
344
   predefinedTests =
345
     testGroup
346
        "predefined tests"
347
        [ testCase "tiny" $
         parseString "2+3" @?= Right (Call "+" [Number 2, Number 3])
349
        , testCase "intro" $ do
350
            act <- parseFile "examples/intro.js"</pre>
351
            exp <- fmap read $ readFile "examples/intro-ast.txt"</pre>
352
            act @?= Right exp
353
   module ParserUtils where
 2
   import SubsAst
3
```

import SubsParser
import Text.Parsec

```
import Text.Parsec.String
  -- for testing parseNumber
  numberParser :: String -> Either ParseError Expr
  numberParser s =
   parse
11
       (do res <- parseNumber
12
           eof
13
          return res)
14
       "ERROR"
15
16
17
  -- for testing parseStr
18
  stringParser :: String -> Either ParseError Expr
19
  stringParser s =
20
    parse
21
       (do res <- parseStr
22
          eof
          return res)
24
       "ERROR"
25
       S
26
27
  -- for testing parseFalse
28
  falseParser :: String -> Either ParseError Expr
  falseParser s =
30
   parse
31
      (do res <- parseFalse
32
33
           eof
          return res)
34
       "ERROR"
35
36
37
  -- for testing parseTrue
38
  trueParser :: String -> Either ParseError Expr
  trueParser s =
40
41
    parse
       (do res <- parseTrue
42
43
          eof
          return res)
44
       "ERROR"
45
46
47
  -- for testing parseUndefined
  undefinedParser :: String -> Either ParseError Expr
```

```
undefinedParser s =
     parse
       (do res <- parseUndefined</pre>
52
           eof
53
           return res)
54
       "ERROR"
55
56
  -- for testing parseAssign
  assignParser :: String -> Either ParseError Expr
59
  assignParser s =
60
    parse
61
       (do res <- parseAssign
62
           eof
63
           return res)
       "ERROR"
65
       S
66
   -- for testing parseCall
68
  callParser :: String -> Either ParseError Expr
69
  callParser s =
    parse
71
       (do res <- parseCall
72
73
           eof
          return res)
74
       "ERROR"
75
76
  -- for testing parseIdent
78
  identParser :: String -> Either ParseError Expr
79
  identParser s =
   parse
81
       (do res <- parseIdent
82
83
           eof
          return res)
84
       "ERROR"
85
86
       S
87
  -- for testing parseParentheses
88
  parenthesesParser :: String -> Either ParseError Expr
89
  parenthesesParser s =
    parse
91
       (do res <- parseParentheses</pre>
92
           eof
```

```
94 return res)
95 "ERROR"
96 s
```