

Haskell intro

Assignment 2

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 26, 2018



Contents

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

1 Design/Implementation

1.1 Choice of Parser combinator Library

We decided to use **Parsec** because of the better error handling capabilities 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 'parseWhitespace' 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 doesn't seem like a nice solution, but we didn't come 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 expressions 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

```
1 module Parser.Impl where
2
3 -- Put your parser implementation in this file (and, if
4   ↳ appropriate,
5 -- in other files in the Parser/ subdirectory)
6 import SubsAst
7 import Text.Parsec.Char
8 import Text.Parsec.Combinator
9 import Text.Parsec.Prim
10 import Text.Parsec.String
11
12 -- ord used in isLegalChar to check if char is in printable ASCII
13   ↳ range
14 import Data.Char
15
16 data ParseError =
17     ParseError String
18     deriving (Eq, Show)
19
20 parseString :: String -> Either ParseError Expr
21 parseString s =
22     case parse
23         (do res <- parseLeadingWhitespace parseExpr
24          eof
25          return res)
26         of
27             "ERROR"
28             s of
29                 Left a -> Left (ParseError (show a))
30                 Right b -> Right b
31
32 posNumber :: Parser Expr
33 posNumber = do
34     n <- many1 digit
35     if length n <= 8
36     then return $ Number $ read n
37     else fail "Number too long"
38
39 negNumber :: Parser Expr
40 negNumber = do
41     m <- string "-"
42     n <- many1 digit
43     if length n <= 8
```

```

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

```

```

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

```

```

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

```

```

173 parseArrayCompr = choice [try parseACBody, parseArrayFor,
    ↳ parseACIf]
174
175 parseACBody :: Parser ArrayCompr
176 parseACBody = do
177     expr <- parseExpr'
178     return (ACBody expr)
179
180 parseACIf :: Parser ArrayCompr
181 parseACIf = do
182     _ <- parseWhitespace (string "if")
183     _ <- parseWhitespace (char '(')
184     expr1 <- parseWhitespace parseExpr'
185     _ <- parseWhitespace (char ')')
186     compr <- parseArrayCompr
187     return (ACIf expr1 compr)
188
189 parseArray :: Parser Expr
190 parseArray = do
191     _ <- parseWhitespace (char '[')
192     exprs <- parseExprs
193     _ <- parseWhitespace (char ']')
194     return (Array exprs)
195
196 -- checks that the char after the backslash is one of the legal
    ↳ possibilites
197 isLegalAfterBackslash :: Char -> Either ParseError Char
198 isLegalAfterBackslash c
199     | c == 'n' = Right '\n'
200     | c == 't' = Right '\t'
201     | c `elem` ['\\', '\n', '\t'] = Right c
202     | otherwise = fail "Backslash followed by invalid char"
203
204 -- extracts the char after the \ to return it together with \
205 isLegalBackslash :: Parser Char
206 isLegalBackslash = do
207     _ <- char '\\'
208     c <- oneOf ['\\', '\n', '\t', '\\']
209     case isLegalAfterBackslash c of
210         Right a -> return a
211         _ -> fail "Fail in Backslash"
212
213 -- checks for printable ascii chars and not \' and \\
214 isLegalChar :: Char -> Bool

```

```

215 isLegalChar c
216   | c == '\\' = False
217   | c == '\\\\' = False
218   | ord c >= 32 && ord c <= 126 = True
219   | otherwise = False
220
221 -- option""(try) checks for newline in string to be skipped
222 -- then checks for backslashes and legal chars
223 parseCharInStr :: Parser Char
224 parseCharInStr = do
225   _ <- option "" (try (string "\\n"))
226   a <- isLegalBackslash <|> satisfy isLegalChar
227   _ <- option "" (try (string "\\n"))
228   return a
229
230 parseStr :: Parser Expr
231 parseStr = do
232   _ <- char '\\'
233   res <- many parseCharInStr
234   _ <- parseWhitespace (char '\\')
235   return (SubsAst.String res)
236
237 parseTrue :: Parser Expr
238 parseTrue = do
239   _ <- string "true"
240   return TrueConst
241
242 parseFalse :: Parser Expr
243 parseFalse = do
244   _ <- string "false"
245   return FalseConst
246
247 parseUndefined :: Parser Expr
248 parseUndefined = do
249   _ <- string "undefined"
250   return Undefined
251
252 parseExpr' :: Parser Expr
253 parseExpr' = parseAdditon `chainl1` parseCompare
254
255 parseCompare :: Parser (Expr -> Expr -> Expr)
256 parseCompare =
257   (do _ <- parseWhitespace (string "<")
258       return (\x y -> Call "<" [x, y])) <|>

```



```

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

```

B Tests

```

1  -- put your tests here, and/or in other files in the tests/
   ↳ directory
2  import Test.Tasty
3  import Test.Tasty.HUnit
4
5  import ParserUtils
6  import SubsAst
7  import SubsParser
8
9  main = defaultMain tests
10
11 tests :: TestTree
12 tests =

```

```

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

```

```

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

```

```

90     ]
91
92     parseTrueTests :: TestTree
93     parseTrueTests =
94         testGroup
95             "parse true"
96             [ testCase "True" $ trueParser ("true") @?= Right (TrueConst)
97             , testCase "True fail" $
98                 show (trueParser ("false")) @?=
99                 "Left \"ERROR\" (line 1, column 1):\nunexpected
100     ↪ \"f\" \nextpecting \"true\""
101     ]
102
103     parseUndefinedTests :: TestTree
104     parseUndefinedTests =
105         testGroup
106             "Undefined"
107             [ testCase "Undefined" $ undefinedParser ("undefined") @?=
108                 ↪ Right (Undefined)
109             , testCase "Undefined fail" $
110                 show (undefinedParser ("defined")) @?=
111                 "Left \"ERROR\" (line 1, column 1):\nunexpected
112     ↪ \"d\" \nextpecting \"undefined\""
113     ]
114
115     parseAssignTests :: TestTree
116     parseAssignTests =
117         testGroup
118             "Assign"
119             [ testCase "Assign" $ assignParser ("x=3") @?= Right (Assign
120     ↪ "x" (Number 3))
121             , testCase "Assign whitespace/special char" $
122                 assignParser ("x = \n 3") @?= Right (Assign "x" (Number 3))
123             , testCase "Assign underline" $
124                 assignParser ("x_x=0") @?= Right (Assign "x_x" (Number 0))
125             ]
126
127     parseCallTests :: TestTree
128     parseCallTests =
129         testGroup
130             "Call"
131             [ testCase "Call" $ callParser ("x(12)") @?= Right (Call "x"
132     ↪ [Number 12])
133             , testCase "Call whitespace" $

```

```

129     callParser ("x ( 12 ) ") @?= Right (Call "x" [Number 12])
130 ]
131
132 parseIdentTests :: TestTree
133 parseIdentTests =
134     testGroup
135         "Ident"
136         [ testCase "Ident" $ identParser ("x_x") @?= Right (Var
137             ↪ "x_x")
138           , testCase "Ident similar to keyword" $
139             identParser ("falsee") @?= Right (Var "falsee")
140           , testCase "Ident keyword" $
141             show (identParser ("false")) @?=
142             ↪ "Left \"ERROR\" (line 1, column 6):\nunexpected end of
143             ↪ input\nexpecting digit, letter or \"_\" \nshould not be a
144             ↪ keyword"
145           , testCase "Ident whitespace" $
146             show (identParser ("x_x   ")) @?=
147             ↪ "Left \"ERROR\" (line 1, column 4):\nunexpected '
148             ↪ '\nexpecting digit, letter, \"_\" or end of input"
149         ]
150
151 parseArrayTests :: TestTree
152 parseArrayTests =
153     testGroup
154         "Array"
155         [ testCase "Array" $
156             parseString ("[1,2]") @?= Right (Array [Number 1, Number
157             ↪ 2])
158           , testCase "Array whitespace" $
159             parseString ("[ 1, 'sds' ] ") @?= Right (Array [Number 1,
160             ↪ String "sds"])
161         ]
162
163 parseStartArrayTests :: TestTree
164 parseStartArrayTests =
165     testGroup
166         "Array Compr"
167         [ testCase "Array for" $
168             parseString ("[for (x of 2) 2]") @?=
169             Right (Compr (ACFor "x" (Number 2) (ACBody (Number 2)))),
170           testCase "Empty Array" $ parseString("[]") @?= Right (Array
171             ↪ [])
172         ]

```

```

166
167 parseParanthesTests :: TestTree
168 parseParanthesTests =
169     testGroup
170         "Parantheses"
171         [ testCase "Parantheses" $ parseString "(1)" @?= Right
172         ↪ (Number 1)
173         , testCase "Parantheses whitespace" $
174           parseString "( 1 )" @?= Right (Number 1)
175         ]
176
177 parseExprs :: TestTree
178 parseExprs =
179     testGroup
180         "parseExprs"
181         [ testCase "parseExprs numbers" $
182           parseString "[1,2,3]" @?= Right (Array [Number 1, Number
183         ↪ 2, Number 3])
184         , testCase "parseExprs" $
185           parseString ("['a','b','c']") @?=
186           Right (Array [String "a", String "b", String "c"])
187         , testCase "parseExprs ident" $
188           parseString ("a (1,2,3)") @?=
189           Right (Call "a" [Number 1, Number 2, Number 3])
190         ]
191
192 parseComma :: TestTree
193 parseComma =
194     testGroup
195         "Comma"
196         [ testCase "Parse Comma" $
197           parseString ("1,2") @=? Right (Comma (Number 1) (Number 2))
198         , testCase "Parse nested commas" $
199           parseString ("1, (1, (3,4))") @?=
200           Right (Comma (Number 1) (Comma (Number 1) (Comma (Number 3)
201         ↪ (Number 4))))
202         , testCase "many commas" $
203           parseString ("1,2,3,'a','b'") @?=
204           Right
205             (Comma
206               (Number 1)
207               (Comma
208                 (Number 2)
209                 (Comma (Number 3) (Comma (String "a") (String
210         ↪ "b"))))))

```

```

207     ]
208
209 parseExprTests :: TestTree
210 parseExprTests =
211     testGroup
212         "parseExpr"
213         [ testCase "Addition" $
214             parseString ("1+1") @=? Right (Call "+" [Number 1, Number
215 → 1])
216         , testCase "Subtraction" $
217             parseString ("1-1") @?= Right (Call "-" [Number 1, Number
218 → 1])
219         , testCase "Mul" $
220             parseString ("1*1") @?= Right (Call "*" [Number 1, Number
221 → 1])
222         , testCase "Mod" $
223             parseString ("1%1") @?= Right (Call "%" [Number 1, Number
224 → 1])
225         , testCase "Smaller Then" $
226             parseString ("1<1") @?= Right (Call "<" [Number 1, Number
227 → 1])
228         , testCase "Equals" $
229             parseString ("1===1") @?= Right (Call "===" [Number 1,
230 → Number 1])
231     ]
232
233 parseArrayCompr :: TestTree
234 parseArrayCompr =
235     testGroup
236         "Array Compr"
237         [ testCase "for" $
238             parseString ("[for (x of 2) 3]") @=?
239             Right (Compr (ACFor "x" (Number 2) (ACBody (Number 3))))
240         , testCase "nested for" $
241             parseString ("[for (x of 2) for (x of 3) 3]") @=?
242             Right
243             (Compr (ACFor "x" (Number 2) (ACFor "x" (Number 3)
244 → (ACBody (Number 3)))))
245         , testCase "nested if" $
246             parseString ("[for (x of 2) if(1) 2]") @=?
247             Right (Compr (ACFor "x" (Number 2) (ACIf (Number 1) (ACBody
248 → (Number 2)))))
249         , testCase "mixed for/if" $
250             parseString ("[for (x of 2) if(1) for (y of 2) if(false)
251 → for(z of 5) 2]") @=?

```

```

243     Right
244     (Compr
245     (ACFor
246     "x"
247     (Number 2)
248     (ACIf
249     (Number 1)
250     (ACFor
251     "y"
252     (Number 2)
253     (ACIf FalseConst (ACFor "z" (Number 5)
  ⇨ (ACBody (Number 2)))))))))
254 ]
255
256 constantTests :: TestTree
257 constantTests =
258   testGroup
259     "constants tests"
260     [ testCase "Number" $ parseString ("2") @?= Right (Number 2)
261     , testCase "String" $ parseString ("'abc'") @?= Right (String
  ⇨ "abc")
262     , testCase "true" $ parseString ("true") @?= Right
  ⇨ (TrueConst)
263     , testCase "false" $ parseString ("false") @?= Right
  ⇨ (FalseConst)
264     , testCase "Undefined" $ parseString ("undefined") @?= Right
  ⇨ (Undefined)
265     , testCase "Ident" $ parseString ("sdsd") @?= Right (Var
  ⇨ "sdsd")
266   ]
267
268 parseSimpleExprTests :: TestTree
269 parseSimpleExprTests =
270   testGroup
271     "Simple expr tests"
272     [ testCase "equal" $
273       parseString ("a===b===c") @?=
274       Right (Call "===" [Call "===" [Var "a", Var "b"], Var "c"])
275     , testCase "assign" $
276       parseString ("a=b=undefined") @?=
277       Right (Assign "a" (Assign "b" Undefined))
278     , testCase "smallerThen" $
279       parseString ("2<3<4") @?=
280       Right (Call "<" [Call "<" [Number 2, Number 3], Number 4])

```



```

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

```

```

323         "*"
324     [ Compr (ACFor "a" (Number 4) (ACBody (Number 1)))
325     , Compr
326         (ACFor
327             "a"
328             (Var "abc")
329             (ACIf
330                 TrueConst
331                 (ACIf FalseConst (ACBody (Call "*" [Number
332     ↪ 2, Number 3]))))
332         ])
333     ]
334
335 parseErrorTest :: TestTree
336 parseErrorTest =
337     testGroup
338         "Parse Fail"
339         [ testCase "let parser fail" $
340             show (parseString "") @=?
341             "Left (ParseError \"\\\"ERROR\\\"\" (line 1, column
342     ↪ 1):\\\"unexpected end of input\\\"nexpecting white space,
343     ↪ \"\\\"//\\\"\", \"\\\"[\\\"\", letter, \"\\\"(\\\"\", digit, \"\\\"-\\\"\",
344     ↪ \"\\\"'\\\"\", \"\\\"true\\\"\", \"\\\"false\\\"\" or
345     ↪ \"\\\"undefined\\\"\\\"\" ) "
346         ]
347
348 predefinedTests :: TestTree
349 predefinedTests =
350     testGroup
351         "predefined tests"
352         [ testCase "tiny" $
353             parseString "2+3" @?= Right (Call "+" [Number 2, Number 3])
354         , testCase "intro" $ do
355             act <- parseFile "examples/intro.js"
356             exp <- fmap read $ readFile "examples/intro-ast.txt"
357             act @?= Right exp
358         ]

```

```

1 module ParserUtils where
2
3 import SubsAst
4 import SubsParser
5 import Text.Parsec

```

```

6  import Text.Parsec.String
7
8  -- for testing parseNumber
9  numberParser :: String -> Either ParseError Expr
10 numberParser s =
11     parse
12         (do res <- parseNumber
13             eof
14             return res)
15         "ERROR"
16         s
17
18 -- for testing parseStr
19 stringParser :: String -> Either ParseError Expr
20 stringParser s =
21     parse
22         (do res <- parseStr
23             eof
24             return res)
25         "ERROR"
26         s
27
28 -- for testing parseFalse
29 falseParser :: String -> Either ParseError Expr
30 falseParser s =
31     parse
32         (do res <- parseFalse
33             eof
34             return res)
35         "ERROR"
36         s
37
38 -- for testing parseTrue
39 trueParser :: String -> Either ParseError Expr
40 trueParser s =
41     parse
42         (do res <- parseTrue
43             eof
44             return res)
45         "ERROR"
46         s
47
48 -- for testing parseUndefined
49 undefinedParser :: String -> Either ParseError Expr

```

```

50 undefinedParser s =
51     parse
52         (do res <- parseUndefined
53             eof
54             return res)
55         "ERROR"
56         s
57
58 -- for testing parseAssign
59 assignParser :: String -> Either ParseError Expr
60 assignParser s =
61     parse
62         (do res <- parseAssign
63             eof
64             return res)
65         "ERROR"
66         s
67
68 -- for testing parseCall
69 callParser :: String -> Either ParseError Expr
70 callParser s =
71     parse
72         (do res <- parseCall
73             eof
74             return res)
75         "ERROR"
76         s
77
78 -- for testing parseIdent
79 identParser :: String -> Either ParseError Expr
80 identParser s =
81     parse
82         (do res <- parseIdent
83             eof
84             return res)
85         "ERROR"
86         s
87
88 -- for testing parseParentheses
89 parenthesesParser :: String -> Either ParseError Expr
90 parenthesesParser s =
91     parse
92         (do res <- parseParentheses
93             eof

```

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