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

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 ‘parseWhitespace‘ which takes care of removing the whitespaces, newlines and other characters in strings as well.

1.3 String parsing

For String parsing we had to check each character to be a printable ASCII[1], which we did via the ordinal of each character. Also for parsing backslashes we had to check that they get escaped and so on, to be able to parse also the \t, \n and so on.

1.4 Precedence and Associativity

We chose to rewrite the grammer so that the operators follow the precedence, by using ‘chainl1‘ and recursive calls through the operators (from === to + and so on). Associativity we left it as described in the assignment.

1.5 Ident and Keywords

We had to check that no Ident has the name of a language keyword (like false or true), this we did by checking an ident against a list with the keywords.

1.6 Usages of try

We had to use try on a few occurences which are the following:

[1] `https://theasciicode.com.ar/ascii-printable-characters/tilde-swung-dash-ascii-code-126.html`

1.6.1 try parseArray

By using try on parseArray we can be sure that it won't be an Array with ArrayFor and therefore don't walk into the issue of an ambiguity so it could be either an Array with Expr or an Array with ArrayFor.

1.6.2 try parseCall

We used a try on parseCall since we have to distinguish if it's an ident by itself or a functiion call (ident + parantheses).

1.6.3 try parseAssign

We had to use try on parseAssign for the sake that ident could have a '=' after it or could be standing alone. This way we are sure if it's either an Assign on ident or an ident itself (Var).

1.6.4 try parseACBody

Since we call parseExpr' here we expect a Expression but it could also end up to be a ACFor or ACIf so we have to try the ACBody to parse the Expression first so that we are sure that there is no for or if there.

1.6.5 (try (string "\\\n"))

Try on newline since it could also fail on strings which don't have any newline, so we have to do a lookahead to be sure if a newline exists or not.

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 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 72 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

```haskell
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

```

```haskell
-- ord used in isLegalChar to check if char is in printable ASCII range
import Data.Char

data ParseError =
  ParseError String
  deriving (Eq, Show)

parseString :: String -> Either ParseError Expr
parseString s =
  case parse
         (do res <- parseLeadingWhitespace parseExpr
             eof
             return res)
         "ERROR"
         s of
    Left a -> Left (ParseError (show a))
    Right b -> Right b

posNumber :: Parser Expr
posNumber = do
  n <- many1 digit
  if length n <= 8
    then return $ Number $ read n
    else fail "Number too long"

negNumber :: Parser Expr
negNumber = do
  m <- string "-"
  n <- many1 digit
  if length n <= 8
    then return $ Number $ read (m ++ n)
    else fail "Number too long"

parseNumber :: Parser Expr
parseNumber = do
  parseWhitespace (posNumber <|> negNumber)

parseParentheses :: Parser Expr
parseParentheses = do
  _ <- parseWhitespace (char '(')
  expr <- parseExpr
  _ <- parseWhitespace (char ')')
  return expr
```

```haskell

parseComment :: Parser ()
parseComment = do
  _ <- string "//"
  _ <- manyTill anyChar (newLine <|> eof)
  return ()

--makes newline be of type ()
newLine :: Parser ()
newLine = do
  _ <- newline
  return ()

parseLeadingWhitespace :: Parser a -> Parser a
parseLeadingWhitespace par = do
  spaces
  optional parseComment
  spaces
  par

parseWhitespace :: Parser a -> Parser a
parseWhitespace par = do
  p <- par
  spaces
  optional parseComment
  spaces
  return p

-- check for comma
parseExpr :: Parser Expr
parseExpr = choice [parseNotComma, parseCons]

parseNotComma :: Parser Expr
parseNotComma = do
  expr1 <- parseWhitespace parseExpr'
  parseComma expr1

parseComma :: Expr -> Parser Expr
parseComma expr1 =
  (do _ <- parseWhitespace (char ',')
      expr2 <- parseWhitespace parseExpr
      return (Comma expr1 expr2)) <|>
  return expr1

```

```haskell
keywords :: [String]
keywords = ["true", "false", "undefined", "for", "of", "if"]

parseCons :: Parser Expr
parseCons =
  choice
    [ try parseArray
    , parseArrayStart
    , try parseCall
    , parseParentheses
    , parseNumber
    , parseStr
    , parseTrue
    , parseFalse
    , parseUndefined
    , try parseAssign
    , parseIdent
    ]

parseIdent :: Parser Expr
parseIdent = do
  fc <- letter
  rest <- many (digit <|> letter <|> char '_')
  let input = fc : rest
  if input `notElem` keywords
    then return (Var input)
    else fail "should not be a keyword"

parseAssign :: Parser Expr
parseAssign = do
  Var ident <- parseWhitespace parseIdent
  _ <- parseWhitespace (char '=')
  expr1 <- parseExpr'
  return (Assign ident expr1)

parseCall :: Parser Expr
parseCall = do
  Var ident <- parseWhitespace parseIdent
  _ <- parseWhitespace (char '(')
  exprs <- parseExprs
  _ <- parseWhitespace (char ')')
  return (Call ident exprs)

parseExprs :: Parser [Expr]
```

```haskell
parseExprs =
  do expr1 <- parseExpr'
     parseCommaExprs expr1
     <|> return []

parseCommaExprs :: Expr -> Parser [Expr]
parseCommaExprs expr1 =
  do _ <- parseWhitespace (char ',')
     expr2 <- parseExprs
     return (expr1 : expr2)
     <|> return [expr1]

parseArrayStart :: Parser Expr
parseArrayStart = do
  _ <- parseWhitespace (char '[')
  compr <- parseArrayFor
  _ <- parseWhitespace (char ']')
  return (Compr compr)

parseArrayFor :: Parser ArrayCompr
parseArrayFor = do
  _ <- parseWhitespace (string "for")
  _ <- parseWhitespace (char '(')
  Var ident <- parseWhitespace parseIdent
  _ <- parseWhitespace (string "of")
  expr1 <- parseWhitespace parseExpr'
  _ <- parseWhitespace (char ')')
  compr <- parseArrayCompr
  return (ACFor ident expr1 compr)

parseArrayCompr :: Parser ArrayCompr
parseArrayCompr = choice [ try parseACBody, parseArrayFor,
↪  parseACIf ]

parseACBody :: Parser ArrayCompr
parseACBody = do
  expr <- parseExpr'
  return (ACBody expr)

parseACIf :: Parser ArrayCompr
parseACIf = do
  _ <- parseWhitespace (string "if")
  _ <- parseWhitespace (char '(')
  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` ['\'', '\\'] = 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"))
```

```haskell
  return a

parseStr :: Parser Expr
parseStr = do
  _ <- char '\''
  res <- many parseCharInStr
  _ <- parseWhitespace (char '\'')
  return (SubsAst.String res)

parseTrue :: Parser Expr
parseTrue = do
  _ <- string "true"
  return TrueConst

parseFalse :: Parser Expr
parseFalse = do
  _ <- string "false"
  return FalseConst

parseUndefined :: Parser Expr
parseUndefined = do
  _ <- string "undefined"
  return Undefined

parseExpr' :: Parser Expr
parseExpr' = parseAdditon `chainl1` parseCompare

parseCompare :: Parser (Expr -> Expr -> Expr)
parseCompare =
  (do _ <- parseWhitespace (string "<")
      return (\x y -> Call "<" [x, y])) <|>
  (do _ <- parseWhitespace (string "===")
      return (\x y -> Call "===" [x, y]))

parseAdditon :: Parser Expr
parseAdditon = do
  prod <- parseWhitespace parseProd
  parseAdditon' prod

parseAdditon' :: Expr -> Parser Expr
parseAdditon' input =
  (do addOp <- parseWhitespace (char '+' <|> char '-')
      cons <- parseProd
      parseAdditon' $ Call [addOp] [input, cons]) <|>
```

```haskell
  return input

parseProd :: Parser Expr
parseProd = do
  cons <- parseWhitespace parseCons
  parseProd' cons

parseProd' :: Expr -> Parser Expr
parseProd' input =
  (do prodOp <- parseWhitespace (char '*' <|> char '%')
      cons <- parseCons
      parseProd' $ Call [prodOp] [input, cons]) <|>
  return input
```

B Tests

```haskell
-- put your tests here, and/or in other files in the tests/
↪  directory
import Test.Tasty
import Test.Tasty.HUnit

import ParserUtils
import SubsAst
import SubsParser

main = defaultMain tests

tests :: TestTree
tests =
  testGroup
    "Tests"
    [ constantTests
    , parseNumberTests
    , parseStringTests
    , parseFalseTests
    , parseTrueTests
    , parseUndefinedTests
    , parseAssignTests
    , parseCallTests
    , parseIdentTests
    , parseArrayTests
    , 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
        ↪  (Number 1)
42      , testCase "Number neg" $ numberParser ("-2") @?= Right
        ↪  (Number (-2))
43      , testCase "Number trailing whitespace" $
44        numberParser ("1      ") @?= Right (Number 1)
45      , testCase "Number 8 long pos" $
46        numberParser ("12345678") @?= Right (Number 12345678)
47      , testCase "Number 8 long neg" $
48        numberParser ("-12345678") @?= Right (Number (-12345678))
49      , testCase "Number too long pos" $
50        show (numberParser ("123456789")) @?=
51        "Left \"ERROR\" (line 1, column 10):\nunexpected end of
          ↪  input\nexpecting digit\nNumber too long"
52      , testCase "Number too long neg" $
53        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 whitespaced" $
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 ("'ü'")) @?=
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
          ↪  \"f\"\nexpecting \"true\""
100     ]
101
```

```haskell
parseUndefinedTests :: TestTree
parseUndefinedTests =
  testGroup
    "Undefined"
    [ testCase "Undefined" $ undefinedParser ("undefined") @?=
    ↪  Right (Undefined)
    , testCase "Undefined fail" $
      show (undefinedParser ("defined")) @?=
      "Left \"ERROR\" (line 1, column 1):\nunexpected
      ↪  \"d\"\nexpecting \"undefined\""
    ]

parseAssignTests :: TestTree
parseAssignTests =
  testGroup
    "Assign"
    [ testCase "Assign" $ assignParser ("x=3") @?= Right (Assign
    ↪  "x" (Number 3))
    , testCase "Assign whitespace/special char" $
      assignParser ("x = \n 3") @?= Right (Assign "x" (Number 3))
    , testCase "Assign underline" $
      assignParser ("x_x=0") @?= Right (Assign "x_x" (Number 0))
    ]

parseCallTests :: TestTree
parseCallTests =
  testGroup
    "Call"
    [ testCase "Call" $ callParser ("x(12)") @?= Right (Call "x"
    ↪  [Number 12])
    , testCase "Call whitespace" $
      callParser ("x ( 12 ) ") @?= Right (Call "x" [Number 12])
    ]

parseIdentTests :: TestTree
parseIdentTests =
  testGroup
    "Ident"
    [ testCase "Ident" $ identParser ("x_x") @?= Right (Var
    ↪  "x_x")
    , testCase "Ident similar to keyword" $
      identParser ("falsee") @?= Right (Var "falsee")
    , testCase "Ident keyword" $
      show (identParser ("false")) @?=
```

```
141         "Left \"ERROR\" (line 1, column 6):\nunexpected end of
            ↪  input\nexpecting digit, letter or \"_\"\nshould not be
            ↪  a keyword"
142       , testCase "Ident whitespace" $
143         show (identParser ("x_x    ")) @?=
144         "Left \"ERROR\" (line 1, column 4):\nunexpected '
            ↪  '\nexpecting digit, letter, \"_\" or end of input"
145       ]
146
147   parseArrayTests :: TestTree
148   parseArrayTests =
149     testGroup
150       "Array"
151       [ testCase "Array" $
152         parseString ("[1,2]") @?= Right (Array [Number 1, Number
            ↪  2])
153       , testCase "Array whitespace" $
154         parseString ("[ 1,  'sds']   ") @?= Right (Array [Number 1,
            ↪  String "sds"])
155       ]
156
157   parseStartArrayTests :: TestTree
158   parseStartArrayTests =
159     testGroup
160       "Array Compr"
161       [ testCase "Array for" $
162         parseString ("[for (x of 2) 2]") @?=
163         Right (Compr (ACFor "x" (Number 2) (ACBody (Number 2)))),
164         testCase "Empty Array" $ parseString("[]") @?= Right(Array
            ↪  [])
165       ]
166
167   parseParanthesTests :: TestTree
168   parseParanthesTests =
169     testGroup
170       "Parantheses"
171       [ testCase "Parantheses" $ parseString ("(1)") @?= Right
          ↪  (Number 1)
172       , testCase "Parantheses whitespace" $
173         parseString ("(   1    )") @?= Right (Number 1)
174       ]
175
176   parseExprs :: TestTree
177   parseExprs =
```

```
178     testGroup
179       "parseExprs"
180       [ testCase "parseExprs numbers" $
181         parseString ("[1,2,3]") @?= Right (Array [Number 1, Number
            ↪  2, Number 3])
182       , testCase "parseExprs" $
183         parseString ("['a','b','c']") @?=
184         Right (Array [String "a", String "b", String "c"])
185       , testCase "parseExprs ident" $
186         parseString ("a (1,2,3)") @?=
187         Right (Call "a" [Number 1, Number 2, Number 3])
188       ]
189
190   parseComma :: TestTree
191   parseComma =
192     testGroup
193       "Comma"
194       [ testCase "Parse Comma" $
195         parseString ("1,2") @?= Right (Comma (Number 1) (Number 2))
196       , testCase "Parse nested commas" $
197         parseString ("1,(1,(3,4))") @?=
198         Right (Comma (Number 1) (Comma (Number 1) (Comma (Number 3)
            ↪  (Number 4))))
199       , testCase "many commas" $
200         parseString ("1,2,3,'a','b'") @?=
201         Right
202           (Comma
203              (Number 1)
204              (Comma
205                 (Number 2)
206                 (Comma (Number 3) (Comma (String "a") (String
                    ↪  "b")))))
207       ]
208
209   parseExprTests :: TestTree
210   parseExprTests =
211     testGroup
212       "parseExpr"
213       [ testCase "Additon" $
214         parseString ("1+1") @?= Right (Call "+" [Number 1, Number
            ↪  1])
215       , testCase "Subtraction" $
216         parseString ("1-1") @?= Right (Call "-" [Number 1, Number
            ↪  1])
```

```
217       , testCase "Mul" $
218         parseString ("1*1") @?= Right (Call "*" [Number 1, Number
            ↪  1])
219       , testCase "Mod" $
220         parseString ("1%1") @?= Right (Call "%" [Number 1, Number
            ↪  1])
221       , testCase "Smaller Then" $
222         parseString ("1<1") @?= Right (Call "<" [Number 1, Number
            ↪  1])
223       , testCase "Equals" $
224         parseString ("1===1") @?= Right (Call "===" [Number 1,
            ↪  Number 1])
225       ]
226
227   parseArrayCompr :: TestTree
228   parseArrayCompr =
229     testGroup
230       "Array Compr"
231       [ testCase "for" $
232         parseString ("[for (x of 2) 3]") @?=
233         Right (Compr (ACFor "x" (Number 2) (ACBody (Number 3))))
234       , testCase "nested for" $
235         parseString ("[for (x of 2) for (x of 3) 3]") @?=
236         Right
237           (Compr (ACFor "x" (Number 2) (ACFor "x" (Number 3)
              ↪  (ACBody (Number 3)))))
238       , testCase "nested if" $
239         parseString ("[for (x of 2) if(1) 2]") @?=
240         Right (Compr (ACFor "x" (Number 2) (ACIf (Number 1) (ACBody
            ↪  (Number 2)))))
241       , testCase "mixed for/if" $
242         parseString ("[for (x of 2) if(1) for (y of 2) if(false)
            ↪  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")
                    ↪  (ACBody (Var "x")))))
301                 (Array [Var "x", Var "y"])))
302       , testCase "correct precedence add" $
303         parseString ("[1,2,3,4] + [1,2,3]") @?=
304         Right
305           (Call
306              "+"
307              [ Array [Number 1, Number 2, Number 3, Number 4]
308              , Array [Number 1, Number 2, Number 3]
309              ])
310       , testCase "precedences" $
311         parseString ("1+2*4-3%4") @?=
312         Right
313           (Call
314              "-"
315              [ Call "+" [Number 1, Call "*" [Number 2, Number 4]]
316              , Call "%" [Number 3, Number 4]
317              ])
318       , testCase "arrayCompr complex" $
319         parseString
320           ("[for (a of 4) 1] * [for (a of abc) if (true) if (false)
              ↪  2*3]") @?=
321         Right
322           (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
                           ↪  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
        ↪  1):\\nunexpected end of input\\nexpecting white space,
        ↪  \\\"//\\\", \\\"[\\\", letter, \\\"(\\\", digit,
        ↪  \\\"-\\\", \\\"'\\\", \\\"true\\\", \\\"false\\\" or
        ↪  \\\"undefined\\\"\")"
342     ]
343
344 predefinedTests :: TestTree
345 predefinedTests =
346   testGroup
347     "predefined tests"
348     [ testCase "tiny" $
349       parseString "2+3" @?= Right (Call "+" [Number 2, Number 3])
350     , testCase "intro" $ do
351         act <- parseFile "examples/intro.js"
352         exp <- fmap read $ readFile "examples/intro-ast.txt"
353         act @?= Right exp
354     ]
```

```
1  module ParserUtils where
2
3  import SubsAst
4  import SubsParser
5  import Text.Parsec.Char
6  import Text.Parsec.Combinator
7  import Text.Parsec.Prim
8  import Text.Parsec.String
9  import Text.Parsec.Error
10
11 -- for testing parseNumber
12 numberParser :: String -> Either ParseError Expr
13 numberParser s =
14   parse
15     (do res <- parseNumber
16         eof
17         return res)
```

```
    "ERROR"
    s

-- for testing parseStr
stringParser :: String -> Either ParseError Expr
stringParser s =
  parse
    (do res <- parseStr
        eof
        return res)
    "ERROR"
    s

-- for testing parseFalse
falseParser :: String -> Either ParseError Expr
falseParser s =
  parse
    (do res <- parseFalse
        eof
        return res)
    "ERROR"
    s

-- for testing parseTrue
trueParser :: String -> Either ParseError Expr
trueParser s =
  parse
    (do res <- parseTrue
        eof
        return res)
    "ERROR"
    s

-- for testing parseUndefined
undefinedParser :: String -> Either ParseError Expr
undefinedParser s =
  parse
    (do res <- parseUndefined
        eof
        return res)
    "ERROR"
    s

-- for testing parseAssign
```

```haskell
assignParser :: String -> Either ParseError Expr
assignParser s =
  parse
    (do res <- parseAssign
        eof
        return res)
    "ERROR"
    s

-- for testing parseCall
callParser :: String -> Either ParseError Expr
callParser s =
  parse
    (do res <- parseCall
        eof
        return res)
    "ERROR"
    s

-- for testing parseIdent
identParser :: String -> Either ParseError Expr
identParser s =
  parse
    (do res <- parseIdent
        eof
        return res)
    "ERROR"
    s

-- for testing parseParentheses
parenthesesParser :: String -> Either ParseError Expr
parenthesesParser s =
  parse
    (do res <- parseParentheses
        eof
        return res)
    "ERROR"
    s
```