

Haskell intro

Assignment 2

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 26, 2018



Contents

1	Design/Implementation	2
1.1	Choice of Parser combinator Library	2
1.2	Whitespace	2
1.3	String parsing	2
1.4	Precedence and Associativity	2
1.5	Ident and Keywords	2
1.6	Usages of try	2
1.6.1	try parseArray	2
1.6.2	try parseCall	3
1.6.3	try parseAssign	3
1.6.4	try parseACBody	3
1.6.5	(try (string "\\n"))	3
2	Code Assessment	3
2.1	Tests	3
2.2	Test Coverage	4
2.3	OnlineTA	4
A	Code Listing	4
B	Tests	11

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.

1.3 String parsing

For String parsing we had to check each character to be a printable ASCII¹, which we did via the ordinal of each character. Also for parsing backslashes we had to check that they get escaped, to be able to parse the `\t`, `\n` and so on.

1.4 Precedence and Associativity

We chose to rewrite the grammar 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 occurrences which are the following:

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 expression or an `Array` with `ArrayFor`.

¹<https://theasciicode.com.ar/ascii-printable-characters/tilde-swung-dash-ascii-code-126.html>

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 function call (ident + parentheses).

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 because it could be a different escaping character.

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 according to the grammar 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' func-

tion 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)

2.3 OnlineTA

We weren’t able to import the interpreter for subscript so the Online TA wasn’t able to run it’s tests on our code.

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  -- ord used in isLegalChar to check if char is in printable ASCII
    ↳ range
12  import Data.Char
13
14  data ParseError =
15      ParseError String
16      deriving (Eq, Show)
17
18  parseString :: String -> Either ParseError Expr
19  parseString s =
20      case parse
21          (do res <- parseLeadingWhitespace parseExpr
22             eof
23             return res)
24          "ERROR"
25      s of
26      Left a -> Left (ParseError (show a))
27      Right b -> Right b
28
29  posNumber :: Parser Expr
30  posNumber = do
31      n <- many1 digit
32      if length n <= 8
33      then return $ Number $ read n
34      else fail "Number too long"
35
36  negNumber :: Parser Expr
37  negNumber = do
38      m <- string "-"
39      n <- many1 digit
40      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     , 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,
174     ↪ parseACIf ]
175
176 parseACBody :: Parser ArrayCompr
177 parseACBody = do
178     expr <- parseExpr'
179     return (ACBody expr)
180
181 parseACIf :: Parser ArrayCompr
182 parseACIf = do
183     _ <- parseWhitespace (string "if")
184     _ <- parseWhitespace (char '(')
185     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
197   ↪ possibilites
198 isLegalAfterBackslash :: Char -> Either ParseError Char
199 isLegalAfterBackslash c
200   | c == 'n' = Right '\n'
201   | c == 't' = Right '\t'
202   | c `elem` ['\\', '\\\\'] = Right c
203   | otherwise = fail "Backslash followed by invalid char"
204
205 -- extracts the char after the \ to return it together with \
206 isLegalBackslash :: Parser Char
207 isLegalBackslash = do
208   _ <- char '\\'
209   c <- oneOf ['\\', 'n', 't', '\\\\']
210   case isLegalAfterBackslash c of
211     Right a -> return a
212     _ -> fail "Fail in Backslash"
213
214 -- checks for printable ascii chars and not \ and \\
215 isLegalChar :: Char -> Bool
216 isLegalChar c
217   | c == '\\' = False
218   | c == '\\\\' = False
219   | ord c >= 32 && ord c <= 126 = True
220   | otherwise = False
221
222 -- option""(try) checks for newline in string to be skipped
223 -- then checks for backslashes and legal chars
224 parseCharInStr :: Parser Char
225 parseCharInStr = do
226   _ <- option "" (try (string "\\n"))
227   a <- isLegalBackslash <|> satisfy isLegalChar
228   _ <- 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 (Parser.Impl)
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")) @?=
57         "Left \"ERROR\" (line 1, column 11):\nunexpected end of
58         ↪ input\nexpecting digit\nNumber too long"
59 ]
60
61 parseStringTests :: TestTree
62 parseStringTests =
63     testGroup
64         "parse string"
65     [ testCase "String" $ stringParser ("abc") @?= Right
66         ↪ (String "abc")
67     , testCase "String alphaNum" $
68         stringParser ("abc123") @?= Right (String "abc123")
69     , 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
71     ↳ asdasd")
72   , testCase "String newline" $
73     stringParser ("'foo\\nbar'") @?= Right (String "foobar")
74   , testCase "Not Allowed ASCII character" $
75     show (stringParser ("'ü'")) @?=
76     "Left \"ERROR\" (line 1, column 2):\nunexpected
77     ↳ '\\252'\nexpecting '\\\\\\\\n', '\\\\\\\\' or '\\'\'"
78   , testCase "backslash chars" $
79     stringParser ("'\\t\\n\\\\\\\\\\\\\\\\'") @?= Right (String
80     ↳ "\\t\n\\")
81   , testCase "string comment" $
82     stringParser ("'//Comment 123'") @?= Right (String
83     ↳ "//Comment 123")
84 ]
85
86 parseFalseTests :: TestTree
87 parseFalseTests =
88   testGroup
89     "parse false"
90     [ testCase "False" $ falseParser ("false") @?= Right
91       ↳ (FalseConst)
92     , testCase "False fail" $
93       show (falseParser ("true")) @?=
94       "Left \"ERROR\" (line 1, column 1):\nunexpected
95       ↳ '\\t'\nexpecting 'false'"
96   ]
97
98 parseTrueTests :: TestTree
99 parseTrueTests =
100   testGroup
101     "parse true"
102     [ testCase "True" $ trueParser ("true") @?= Right (TrueConst)
103     , testCase "True fail" $
104       show (trueParser ("false")) @?=
105       "Left \"ERROR\" (line 1, column 1):\nunexpected
106       ↳ '\\f'\nexpecting 'true'"
107   ]

```

```

102 parseUndefinedTests :: TestTree
103 parseUndefinedTests =
104     testGroup
105         "Undefined"
106     [ testCase "Undefined" $ undefinedParser ("undefined") @?=
107         ↳ Right (Undefined)
108     , testCase "Undefined fail" $
109         show (undefinedParser ("defined")) @?=
110         "Left \"ERROR\" (line 1, column 1):\nunexpected
111         ↳ \"d\" \nextpecting \"undefined\""
112     ]
113
114 parseAssignTests :: TestTree
115 parseAssignTests =
116     testGroup
117         "Assign"
118     [ testCase "Assign" $ assignParser ("x=3") @?= Right (Assign
119         ↳ "x" (Number 3))
120     , testCase "Assign whitespace/special char" $
121         assignParser ("x = \n 3") @?= Right (Assign "x" (Number 3))
122     , testCase "Assign underline" $
123         assignParser ("x_x=0") @?= Right (Assign "x_x" (Number 0))
124     ]
125
126 parseCallTests :: TestTree
127 parseCallTests =
128     testGroup
129         "Call"
130     [ testCase "Call" $ callParser ("x(12)") @?= Right (Call "x"
131         ↳ [Number 12])
132     , testCase "Call whitespace" $
133         callParser ("x ( 12 ) ") @?= Right (Call "x" [Number 12])
134     ]
135
136 parseIdentTests :: TestTree
137 parseIdentTests =
138     testGroup
139         "Ident"
140     [ testCase "Ident" $ identParser ("x_x") @?= Right (Var
141         ↳ "x_x")
142     , testCase "Ident similar to keyword" $
143         identParser ("falsee") @?= Right (Var "falsee")
144     , testCase "Ident keyword" $
145         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
182             ↪ 2, Number 3])
183     , testCase "parseExprs" $
184         parseString ("['a','b','c']") @?=
185         Right (Array [String "a", String "b", String "c"])
186     , testCase "parseExprs ident" $
187         parseString ("a (1,2,3)") @?=
188         Right (Call "a" [Number 1, Number 2, Number 3])
189     ]
190
191 parseComma :: TestTree
192 parseComma =
193     testGroup
194         "Comma"
195     [ testCase "Parse Comma" $
196         parseString ("1,2") @?= Right (Comma (Number 1) (Number 2))
197     , testCase "Parse nested commas" $
198         parseString ("1,(1,(3,4))") @?=
199         Right (Comma (Number 1) (Comma (Number 1) (Comma (Number 3)
200             ↪ (Number 4))))
201     , testCase "many commas" $
202         parseString ("1,2,3,'a','b'") @?=
203         Right
204             (Comma
205                 (Number 1)
206                 (Comma
207                     (Number 2)
208                     (Comma (Number 3) (Comma (String "a") (String
209                         ↪ "b")))))
210     ]
211
212 parseExprTests :: TestTree
213 parseExprTests =
214     testGroup
215         "parseExpr"
216     [ testCase "Addition" $
217         parseString ("1+1") @?= Right (Call "+" [Number 1, Number
218             ↪ 1])
219     , testCase "Subtraction" $
220         parseString ("1-1") @?= Right (Call "-" [Number 1, Number
221             ↪ 1])

```



```

217     , testCase "Mul" $
218       parseString ("1*1") @?= Right (Call "*" [Number 1, Number
219         ↪ 1])
219     , testCase "Mod" $
220       parseString ("1%1") @?= Right (Call "%" [Number 1, Number
221         ↪ 1])
221     , testCase "Smaller Than" $
222       parseString ("1<1") @?= Right (Call "<" [Number 1, Number
223         ↪ 1])
223     , testCase "Equals" $
224       parseString ("1==1") @?= Right (Call "==" [Number 1,
225         ↪ 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)
238             ↪ (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
241           ↪ (Number 2)))))
241     , testCase "mixed for/if" $
242       parseString ("[for (x of 2) if(1) for (y of 2) if(false)
243         ↪ for(z of 5) 2]") @?=
244         Right
245           (Compr
246             (ACFor
247               "x"
248               (Number 2)
249               (ACIf
250                 (Number 1)
251                 (ACFor
252                   "y"
253                   (Number 2)
254                   (ACIf FalseConst (ACFor "z" (Number 5)
255                     ↪ (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
262             ↪ "abc")
263         , testCase "true" $ parseString ("true") @?= Right
264             ↪ (TrueConst)
265         , testCase "false" $ parseString ("false") @?= Right
266             ↪ (FalseConst)
267         , testCase "Undefined" $ parseString ("undefined") @?= Right
268             ↪ (Undefined)
269         , testCase "Ident" $ parseString ("sdasd") @?= Right (Var
270             ↪ "sdasd")
271     ]
272
273 parseSimpleExprTests :: TestTree
274 parseSimpleExprTests =
275     testGroup
276         "Simple expr tests"
277         [ testCase "equal" $
278             parseString ("a===b===c") @?=
279             Right (Call "===" [Call "===" [Var "a", Var "b"], Var "c"])
280         , testCase "assign" $
281             parseString ("a=b=undefined") @?=
282             Right (Assign "a" (Assign "b" Undefined))
283         , testCase "smallerThen" $
284             parseString ("2<3<4") @?=
285             Right (Call "<" [Call "<" [Number 2, Number 3], Number 4])
286         , testCase "whitespace" $
287             parseString ("12 \v \t \t \n") @?= Right (Number 12)
288         , testCase "comment" $
289             parseString ("1 //comment 11212121212\n,2") @?=
290             Right (Comma (Number 1) (Number 2))
291         , testCase "comment at start" $
292             parseString ("//comment \n 2 ") @?= Right (Number 2)
293     ]
294
295 parseComplexExprTests :: TestTree
296 parseComplexExprTests =
297     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
325       "*"
326       [ Compr (ACFor "a" (Number 4) (ACBody (Number 1)))
327         , Compr
328           (ACFor
329             "a"
330             (Var "abc")
331             (ACIf
332               TrueConst
333               (ACIf FalseConst (ACBody (Call "*" [Number

```

```

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)

```

```

18     "ERROR"
19     s
20
21 -- for testing parseStr
22 stringParser :: String -> Either ParseError Expr
23 stringParser s =
24     parse
25         (do res <- parseStr
26             eof
27             return res)
28     "ERROR"
29     s
30
31 -- for testing parseFalse
32 falseParser :: String -> Either ParseError Expr
33 falseParser s =
34     parse
35         (do res <- parseFalse
36             eof
37             return res)
38     "ERROR"
39     s
40
41 -- for testing parseTrue
42 trueParser :: String -> Either ParseError Expr
43 trueParser s =
44     parse
45         (do res <- parseTrue
46             eof
47             return res)
48     "ERROR"
49     s
50
51 -- for testing parseUndefined
52 undefinedParser :: String -> Either ParseError Expr
53 undefinedParser s =
54     parse
55         (do res <- parseUndefined
56             eof
57             return res)
58     "ERROR"
59     s
60
61 -- for testing parseAssign

```

```

62 assignParser :: String -> Either ParseError Expr
63 assignParser s =
64     parse
65         (do res <- parseAssign
66             eof
67             return res)
68         "ERROR"
69         s
70
71 -- for testing parseCall
72 callParser :: String -> Either ParseError Expr
73 callParser s =
74     parse
75         (do res <- parseCall
76             eof
77             return res)
78         "ERROR"
79         s
80
81 -- for testing parseIdent
82 identParser :: String -> Either ParseError Expr
83 identParser s =
84     parse
85         (do res <- parseIdent
86             eof
87             return res)
88         "ERROR"
89         s
90
91 -- for testing parseParentheses
92 parenthesesParser :: String -> Either ParseError Expr
93 parenthesesParser s =
94     parse
95         (do res <- parseParentheses
96             eof
97             return res)
98         "ERROR"
99         s

```
