Haskell intro

Assignment 2

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 26, 2018



Contents

1	Desi	Design/Implementation															2						
	1.1	1 Choice of Parser combinator Library														2							
	1.2	.2 Whitespace												2									
	1.3	-											2										
	1.4	.4 Precedence and Associativity												2									
	1.5 Ident and Keywords													2									
	1.6	Usages of try														2							
		1.6.1		y pars																			2
		1.6.2	try	y pars	seCa	11.																	3
		1.6.3	try	y pars	seAs	sign	١.																3
		1.6.4	-	y pars		_																	3
		1.6.5		ry (sti																			3
2	Code Assessment															3							
	2.1	2.1 Tests												3									
	2.2												4										
	2.3	Online	eTA																				4
A	Cod	e Listin	ıg																				4
В	Test	S																					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.

Inttps://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 " $\setminus \setminus n$ "))

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

```
-- ord used in isLegalChar to check if char is in printable ASCII
   import Data.Char
13
   data ParseError =
     ParseError String
15
     deriving (Eq, Show)
16
17
   parseString :: String -> Either ParseError Expr
   parseString s =
19
20
     case parse
             (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
     n <- many1 digit
31
     if length n <= 8</pre>
32
       then return $ Number $ read n
       else fail "Number too long"
34
35
   negNumber :: Parser Expr
36
   negNumber = do
     m <- string "-"
38
     n <- many1 digit
39
     if length n <= 8</pre>
       then return $ Number $ read (m ++ n)
41
       else fail "Number too long"
42
   parseNumber :: Parser Expr
44
   parseNumber = do
45
     parseWhitespace (posNumber <|> negNumber)
47
  parseParentheses :: Parser Expr
48
   parseParentheses = do
49
     _ <- parseWhitespace (char '(')</pre>
50
     expr <- parseExpr</pre>
51
     _ <- parseWhitespace (char ')')</pre>
     return expr
```

```
54
  parseComment :: Parser ()
   parseComment = do
     _ <- string "//"
57
     _ <- manyTill anyChar (newLine <|> eof)
58
    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
     spaces
71
72
     par
73
  parseWhitespace :: Parser a -> Parser a
  parseWhitespace par = do
75
    p <- par
76
     spaces
77
     optional parseComment
78
     spaces
79
    return p
80
81
  -- check for comma
82
83 parseExpr :: Parser Expr
   parseExpr = choice [parseNotComma, parseCons]
84
85
  parseNotComma :: Parser Expr
86
  parseNotComma = do
     expr1 <- parseWhitespace parseExpr'</pre>
88
     parseComma expr1
89
  parseComma :: Expr -> Parser Expr
91
  parseComma expr1 =
92
     (do _ <- parseWhitespace (char ',')</pre>
93
         expr2 <- parseWhitespace parseExpr</pre>
94
         return (Comma expr1 expr2)) <|>
95
     return expr1
96
97
```

```
keywords :: [String]
   keywords = ["true", "false", "undefined", "for", "of", "if"]
100
   parseCons :: Parser Expr
101
102
   parseCons =
      choice
103
        [ try parseArray
104
        , parseArrayStart
105
        , try parseCall
106
        , parseParentheses
107
108
        , parseNumber
        , parseStr
109
        , parseTrue
110
        , parseFalse
111
        , parseUndefined
112
        , try parseAssign
113
        , parseIdent
114
        1
115
116
   parseIdent :: Parser Expr
117
   parseIdent = do
118
     fc <- letter
119
      rest <- many (digit <|> letter <|> char '_')
120
      let input = fc : rest
121
      if input `notElem` keywords
122
        then return (Var input)
123
        else fail "should not be a keyword"
124
   parseAssign :: Parser Expr
126
   parseAssign = do
127
      Var ident <- parseWhitespace parseIdent</pre>
      _ <- parseWhitespace (char '=')</pre>
129
      expr1 <- parseExpr'</pre>
130
     return (Assign ident expr1)
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
  parseExprs :: Parser [Expr]
```

```
parseExprs =
142
      do expr1 <- parseExpr'</pre>
143
144
         parseCommaExprs expr1
         <|> return []
145
146
    parseCommaExprs :: Expr -> Parser [Expr]
147
    parseCommaExprs expr1 =
148
      do _ <- parseWhitespace (char ',')</pre>
149
150
         expr2 <- parseExprs</pre>
         return (expr1 : expr2)
151
         <|> return [expr1]
152
153
    parseArrayStart :: Parser Expr
154
    parseArrayStart = do
155
      _ <- parseWhitespace (char '[')</pre>
156
      compr <- parseArrayFor</pre>
157
      _ <- parseWhitespace (char ']')</pre>
158
      return (Compr compr)
159
    parseArrayFor :: Parser ArrayCompr
161
    parseArrayFor = do
162
      _ <- parseWhitespace (string "for")</pre>
163
      _ <- parseWhitespace (char '(')</pre>
164
      Var ident <- parseWhitespace parseIdent</pre>
165
      _ <- parseWhitespace (string "of")</pre>
      expr1 <- parseWhitespace parseExpr'</pre>
167
      _ <- parseWhitespace (char ')')</pre>
168
169
      compr <- parseArrayCompr</pre>
      return (ACFor ident exprl compr)
170
171
    parseArrayCompr :: Parser ArrayCompr
172
    parseArrayCompr = choice [ try parseACBody, parseArrayFor,
173
    \rightarrow parseACIf ]
174
    parseACBody :: Parser ArrayCompr
175
    parseACBody = do
176
      expr <- parseExpr'</pre>
177
178
      return (ACBody expr)
179
   parseACIf :: Parser ArrayCompr
180
    parseACIf = do
      _ <- parseWhitespace (string "if")</pre>
182
      _ <- parseWhitespace (char '(')</pre>
183
      expr1 <- parseWhitespace parseExpr'
```

```
_ <- parseWhitespace (char ')')</pre>
185
      compr <- parseArrayCompr</pre>
186
      return (ACIf expr1 compr)
187
188
   parseArray :: Parser Expr
189
   parseArray = do
      _ <- parseWhitespace (char '[')</pre>
191
     exprs <- parseExprs
192
      _ <- parseWhitespace (char ']')</pre>
193
     return (Array exprs)
194
195
   -- checks that the char after the backslash is one of the legal
    \rightarrow 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 \
204
   isLegalBackslash :: Parser Char
205
   isLegalBackslash = do
206
     _ <- char '\\'
207
      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
214
   isLegalChar c
215
     | c == '\'' = False
216
      | c == '\\' = False
     | ord c >= 32 && ord c <= 126 = True
     | otherwise = False
219
   -- option""(try) checks for newline in string to be skipped
221
   -- then checks for backslashes and legal chars
222
   parseCharInStr :: Parser Char
223
   parseCharInStr = do
     _ <- option "" (try (string "\\\n"))</pre>
225
     a <- isLegalBackslash <|> satisfy isLegalChar
226
     \_ <- option "" (try (string "\\\n"))
```

```
228
     return a
   parseStr :: Parser Expr
230
   parseStr = do
231
232
      _ <- char '\''
     res <- many parseCharInStr
233
      _ <- parseWhitespace (char '\'')</pre>
234
      return (SubsAst.String res)
235
236
   parseTrue :: Parser Expr
237
   parseTrue = do
238
      _ <- string "true"</pre>
     return TrueConst
240
241
   parseFalse :: Parser Expr
242
   parseFalse = do
243
     _ <- string "false"</pre>
244
    return FalseConst
245
   parseUndefined :: Parser Expr
247
   parseUndefined = do
248
     _ <- string "undefined"</pre>
249
     return Undefined
250
251
   parseExpr' :: Parser Expr
252
   parseExpr' = parseAdditon `chainl1` parseCompare
253
254
   parseCompare :: Parser (Expr -> Expr -> Expr)
   parseCompare =
256
      (do \_ <- parseWhitespace (string "<")
257
          return (x y -> Call "<" [x, y])) <|>
      (do _ <- parseWhitespace (string "===")</pre>
259
         return (\x y -> Call "===" [x, y]))
260
261
   parseAdditon :: Parser Expr
262
   parseAdditon = do
263
      prod <- parseWhitespace parseProd</pre>
264
      parseAdditon' prod
266
   parseAdditon' :: Expr -> Parser Expr
267
   parseAdditon' input =
      (do addOp <- parseWhitespace (char '+' <|> char '-')
269
          cons <- parseProd
270
          parseAdditon' $ Call [addOp] [input, cons]) <|>
```

```
return input
272
273
    parseProd :: Parser Expr
274
    parseProd = do
275
      cons <- parseWhitespace parseCons</pre>
276
      parseProd' cons
277
278
    parseProd' :: Expr -> Parser Expr
279
    parseProd' input =
280
      (do prodOp <- parseWhitespace (char '*' <|> char '%')
281
282
          cons <- parseCons</pre>
          parseProd' $ Call [prodOp] [input, cons]) <|>
283
      return input
284
```

B Tests

```
-- put your tests here, and/or in other files in the tests/
   \rightarrow directory
   import Test.Tasty
   import Test.Tasty.HUnit
  import ParserUtils
   import SubsAst
   import SubsParser (Parser.Impl)
   main = defaultMain tests
  tests :: TestTree
11
   tests =
12
     testGroup
13
       "Tests"
14
       [ constantTests
15
       , parseNumberTests
16
       , parseStringTests
17
       , parseFalseTests
18
       , parseTrueTests
19
20
       , parseUndefinedTests
       , parseAssignTests
21
       , parseCallTests
22
       , parseIdentTests
23
       , parseArrayTests
24
       , parseStartArrayTests
25
```

```
, parseParanthesTests
26
       , parseExprs
       , 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"
40
       [ testCase "Number pos" $ numberParser ("1") @?= Right
41
          (Number 1)
       , testCase "Number neg" $ numberParser ("-2") @?= Right
        \hookrightarrow (Number (-2))
       , testCase "Number trailing whitespace" $
43
         numberParser ("1
                               ") @?= Right (Number 1)
       , testCase "Number 8 long pos" $
45
         numberParser ("12345678") @?= Right (Number 12345678)
46
       , testCase "Number 8 long neg" $
47
         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")) @?=
         "Left \"ERROR\" (line 1, column 11):\nunexpected end of
54
          → input\nexpecting digit\nNumber too long"
56
   parseStringTests :: TestTree
57
   parseStringTests =
59
     testGroup
       "parse string"
60
       [ testCase "String" $ stringParser ("'abc'") @?= Right
61
       , testCase "String alphaNum" $
62
         stringParser ("'abc123'") @?= Right (String "abc123")
63
       , testCase "String allowed special chars" $
```

```
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
70
          → asdasd")
        , testCase "String newline" $
71
          stringParser ("'foo\\\nbar'") @?= Right (String "foobar")
72
        , testCase "Not Allowed ASCII character" $
73
          show (stringParser ("'ü'")) @?=
74
          "Left \"ERROR\" (line 1, column 2):\nunexpected
75
          → \"\\252\"\nexpecting \"\\\\\n\", \"\\\\\" or \"'\\""
        , testCase "backslash chars" $
76
          stringParser ("'\\t\\n\\'\\\") @?= Right (String
77
          \rightarrow "\t\n'\\")
        , testCase "string comment" $
78
          stringParser ("'//Comment 123'") @?= Right (String
          \rightarrow "//Comment 123")
        1
80
81
   parseFalseTests :: TestTree
82
   parseFalseTests =
83
     testGroup
84
        "parse false"
85
        [ testCase "False" $ falseParser ("false") @?= Right
86
        → (FalseConst)
        , testCase "False fail" $
          show (falseParser ("true")) @?=
88
          "Left \"ERROR\" (line 1, column 1):\nunexpected
89
          \rightarrow \"t\"\nexpecting \"false\""
        1
90
91
92
   parseTrueTests :: TestTree
   parseTrueTests =
93
     testGroup
94
        "parse true"
        [ testCase "True" $ trueParser ("true") @?= Right (TrueConst)
96
        , testCase "True fail" $
97
          show (trueParser ("false")) @?=
98
          "Left \"ERROR\" (line 1, column 1):\nunexpected
          → \"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
     testGroup
114
        "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
        1
121
122
   parseCallTests :: TestTree
123
   parseCallTests =
124
     testGroup
125
        "Call"
        [ testCase "Call" $ callParser ("x(12)") @?= Right (Call "x"
127
        → [Number 12])
        , testCase "Call whitespace" $
128
          callParser ("x ( 12 ) ") @?= Right (Call "x" [Number 12])
129
        1
130
   parseIdentTests :: TestTree
132
   parseIdentTests =
133
      testGroup
        "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 '
          → '\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
          \hookrightarrow 2])
        , testCase "Array whitespace" $
153
          parseString ("[ 1, 'sds'] ") @?= Right (Array [Number 1,
154

    String "sds"])

        1
155
156
   parseStartArrayTests :: TestTree
157
   parseStartArrayTests =
158
     testGroup
159
        "Array Compr"
160
        [ testCase "Array for" $
161
          parseString ("[for (x of 2) 2]") @?=
162
          Right (Compr (ACFor "x" (Number 2) (ACBody (Number 2)))),
163
          testCase "Empty Array" $ parseString("[]") @?= Right(Array
164
          → [])
165
166
   parseParanthesTests :: TestTree
167
168
   parseParanthesTests =
     testGroup
169
        "Parantheses"
170
        [ testCase "Parantheses" $ parseString ("(1)") @?= Right
        → (Number 1)
        , testCase "Parantheses whitespace" $
172
          parseString ("( 1 )") @?= Right (Number 1)
173
174
175
   parseExprs :: TestTree
176
   parseExprs =
```

```
testGroup
178
        "parseExprs"
179
        [ testCase "parseExprs numbers" $
180
          parseString ("[1,2,3]") @?= Right (Array [Number 1, Number
181
           , 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
    parseComma =
191
      testGroup
192
        "Comma"
193
        [ testCase "Parse Comma" $
194
          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
              (Number 4))))
        , testCase "many commas" $
199
          parseString ("1,2,3,'a','b'") @?=
200
          Right
201
             (Comma
202
203
                (Number 1)
                (Comma
204
                    (Number 2)
205
                    (Comma (Number 3) (Comma (String "a") (String
206
                    → "b")))))
        1
207
208
   parseExprTests :: TestTree
209
   parseExprTests =
210
      testGroup
211
212
        "parseExpr"
        [ testCase "Additon" $
213
          parseString ("1+1") @?= Right (Call "+" [Number 1, Number
214
           \hookrightarrow 1])
        , testCase "Subtraction" $
215
          parseString ("1-1") @?= Right (Call "-" [Number 1, Number
216
           \hookrightarrow 1])
```

```
, testCase "Mul" $
217
          parseString ("1*1") @?= Right (Call "*" [Number 1, Number
218
           \hookrightarrow 1])
        , 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 1])
        , testCase "Equals" $
223
          parseString ("1===1") @?= Right (Call "===" [Number 1,
224
           → Number 1])
        1
225
226
   parseArrayCompr :: TestTree
227
   parseArrayCompr =
228
      testGroup
229
        "Array Compr"
230
        [ testCase "for" $
231
          parseString ("[for (x \text{ of } 2) \text{ 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
             , testCase "nested if" $
238
          parseString ("[for (x of 2) if(1) 2]") @?=
239
          Right (Compr (ACFor "x" (Number 2) (ACIf (Number 1) (ACBody
240
           , testCase "mixed for/if" $
241
          parseString ("[for (x of 2) if(1) for (y of 2) if(false)
242
           \rightarrow for(z of 5) 2]") @?=
          Right
243
             (Compr
244
                (ACFor
245
                    " X "
246
247
                    (Number 2)
                    (ACIf
248
                       (Number 1)
249
                       (ACFor
250
                          " y "
251
                           (Number 2)
252
                           (ACIf FalseConst (ACFor "z" (Number 5)
253
                             (ACBody (Number 2)))))))
```

```
254
        ]
255
   constantTests :: TestTree
256
   constantTests =
257
     testGroup
258
        "constants tests"
259
        [ 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
        , testCase "Undefined" $ parseString ("undefined") @?= Right
264
        \hookrightarrow (Undefined)
        , testCase "Ident" $ parseString ("sdsd") @?= Right (Var
265
           "sdsd")
        ]
266
267
   parseSimpleExprTests :: TestTree
268
   parseSimpleExprTests =
269
     testGroup
270
        "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
          Right (Call "<" [Call "<" [Number 2, Number 3], Number 4])</pre>
280
        , testCase "whitespace" $
281
          parseString ("12
                              \v \t\t
                                          \n") @?= Right (Number 12)
282
        , testCase "comment" $
283
          parseString ("1 //comment 11212121212\n,2") @?=
284
          Right (Comma (Number 1) (Number 2))
285
        , testCase "comment at start" $
          parseString ("//comment \n 2
                                          ") @?= Right (Number 2)
287
288
        ]
289
   parseComplexExprTests :: TestTree
290
   parseComplexExprTests =
291
292
      testGroup
```

```
"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
                   (Array [Var "x", Var "y"])))
301
302
        , testCase "correct precedence add" $
          parseString ("[1,2,3,4] + [1,2,3]") @?=
303
         Right
304
            (Call
305
               \pi + \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
               1)
317
        , testCase "arrayCompr complex" $
318
          parseString
319
            ("[for (a of 4) 1] * [for (a of abc) if (true) if (false)
320
            Right
321
            (Call
322
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
                          ])
332
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,
           \rightarrow \\\"/\\\", \\\"[\\\", letter, \\\"(\\\", digit,
           → \\\"-\\\", \\\"'\\\", \\\"true\\\", \\\"false\\\" or
           → \\\"undefined\\\"\")"
        1
342
343
   predefinedTests :: TestTree
344
   predefinedTests =
345
     testGroup
346
        "predefined tests"
        [ testCase "tiny" $
348
          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
        ]
354
   module ParserUtils where
```

```
import SubsAst
3
  import SubsParser
   import Text.Parsec.Char
   import Text.Parsec.Combinator
   import Text.Parsec.Prim
   import Text.Parsec.String
   import Text.Parsec.Error
9
10
   -- for testing parseNumber
  numberParser :: String -> Either ParseError Expr
12
   numberParser s =
13
     parse
14
       (do res <- parseNumber
15
           eof
16
           return res)
```

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

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