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 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 ASCII[1], 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‘ 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)

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

```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
```

```

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

```

```
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]
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'
```

```haskell
  _ <- parseWhitespace (char ')')
  compr <- parseArrayCompr
  return (ACIf expr1 compr)

parseArray :: Parser Expr
parseArray = do
  _ <- parseWhitespace (char '[')
  exprs <- parseExprs
  _ <- parseWhitespace (char ']')
  return (Array exprs)

-- checks that the char after the backslash is one of the legal possibilites
isLegalAfterBackslash :: Char -> Either ParseError Char
isLegalAfterBackslash c
  | c == 'n' = Right '\n'
  | c == 't' = Right '\t'
  | c `elem` ['\'', '\\'] = Right c
  | otherwise = fail "Backslash followed by invalid char"

-- extracts the char after the \ to return it together with \
isLegalBackslash :: Parser Char
isLegalBackslash = do
  _ <- char '\\'
  c <- oneOf ['\'', 'n', 't', '\\']
  case isLegalAfterBackslash c of
    Right a -> return a
    _ -> fail "Fail in Backslash"

-- checks for printable ascii chars and not \' and \\
isLegalChar :: Char -> Bool
isLegalChar c
  | c == '\'' = False
  | c == '\\' = False
  | ord c >= 32 && ord c <= 126 = True
  | otherwise = False

-- option""(try) checks for newline in string to be skipped
-- then checks for backslashes and legal chars
parseCharInStr :: Parser Char
parseCharInStr = do
  _ <- option "" (try (string "\\\n"))
  a <- isLegalBackslash <|> satisfy isLegalChar
  _ <- option "" (try (string "\\\n"))
  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 (Parser.Impl)

main = defaultMain tests

tests :: TestTree
tests =
  testGroup
    "Tests"
    [ constantTests
    , parseNumberTests
    , parseStringTests
    , parseFalseTests
    , parseTrueTests
    , parseUndefinedTests
    , parseAssignTests
    , parseCallTests
    , parseIdentTests
    , parseArrayTests
    , parseStartArrayTests
    , parseParanthesTests
    , parseExprs
    , parseComma
    , parseExprTests
    , parseArrayCompr
    , parseSimpleExprTests
    , parseComplexExprTests
    , predefinedTests
    , parseErrorTest
    ]

parseNumberTests :: TestTree
parseNumberTests =
  testGroup
    "parse number"
    [ testCase "Number pos" $ numberParser ("1") @?= Right (Number 1)
    , testCase "Number neg" $ numberParser ("-2") @?= Right (Number (-2))
    , testCase "Number trailing whitespace" $
      numberParser ("1     ") @?= Right (Number 1)
    , testCase "Number 8 long pos" $
      numberParser ("12345678") @?= Right (Number 12345678)
    , testCase "Number 8 long neg" $
      numberParser ("-12345678") @?= Right (Number (-12345678))
    , testCase "Number too long pos" $
      show (numberParser ("123456789")) @?=
      "Left \"ERROR\" (line 1, column 10):\nunexpected end of input\nexpecting digit\nNumber too long"
    , testCase "Number too long neg" $
      show (numberParser ("-123456789")) @?=
      "Left \"ERROR\" (line 1, column 11):\nunexpected end of input\nexpecting digit\nNumber too long"
    ]

parseStringTests :: TestTree
parseStringTests =
  testGroup
    "parse string"
    [ testCase "String" $ stringParser ("'abc'") @?= Right (String "abc")
    , testCase "String alphaNum" $
      stringParser ("'abc123'") @?= Right (String "abc123")
    , testCase "String allowed special chars" $
```

```
      stringParser ("'abc\\n\\t'") @?= Right (String "abc\n\t")
    , testCase "String  not allowed special char" $
      show (stringParser ("'\\\a'")) @?=
      "Left \"ERROR\" (line 1, column 3):\nunexpected \"\\a\""
    , testCase "String whitespaced" $
      stringParser ("'asdas asdasd'") @?= Right (String "asdas
       ↪  asdasd")
    , testCase "String newline" $
      stringParser ("'foo\\\nbar'") @?= Right (String "foobar")
    , testCase "Not Allowed ASCII character" $
      show (stringParser ("'ü'")) @?=
      "Left \"ERROR\" (line 1, column 2):\nunexpected
       ↪  \"\\252\"\nexpecting \"\\\\\\n\", \"\\\\\" or \"'\""
    , testCase "backslash chars" $
      stringParser ("'\\t\\n\\'\\\\\'") @?= Right (String
       ↪  "\t\n'\\")
    , testCase "string comment" $
      stringParser ("'//Comment 123'") @?= Right (String
       ↪  "//Comment 123")
    ]

parseFalseTests :: TestTree
parseFalseTests =
  testGroup
    "parse false"
    [ testCase "False" $ falseParser ("false") @?= Right
     ↪  (FalseConst)
    , testCase "False fail" $
      show (falseParser ("true")) @?=
      "Left \"ERROR\" (line 1, column 1):\nunexpected
       ↪  \"t\"\nexpecting \"false\""
    ]

parseTrueTests :: TestTree
parseTrueTests =
  testGroup
    "parse true"
    [ testCase "True" $ trueParser ("true") @?= Right (TrueConst)
    , testCase "True fail" $
      show (trueParser ("false")) @?=
      "Left \"ERROR\" (line 1, column 1):\nunexpected
       ↪  \"f\"\nexpecting \"true\""
    ]

```

```
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")) @?=
      "Left \"ERROR\" (line 1, column 6):\nunexpected end of input\nexpecting digit, letter or \"_\"\nshould not be a keyword"
    , testCase "Ident whitespace" $
      show (identParser ("x_x     ")) @?=
      "Left \"ERROR\" (line 1, column 4):\nunexpected ' '\nexpecting digit, letter, \"_\" or end of input"
    ]

parseArrayTests :: TestTree
parseArrayTests =
  testGroup
    "Array"
    [ testCase "Array" $
      parseString ("[1,2]") @?= Right (Array [Number 1, Number 2])
    , testCase "Array whitespace" $
      parseString ("[ 1,  'sds']    ") @?= Right (Array [Number 1, String "sds"])
    ]

parseStartArrayTests :: TestTree
parseStartArrayTests =
  testGroup
    "Array Compr"
    [ testCase "Array for" $
      parseString ("[for (x of 2) 2]") @?=
      Right (Compr (ACFor "x" (Number 2) (ACBody (Number 2)))),
      testCase "Empty Array" $ parseString("[]") @?= Right(Array [])
    ]

parseParanthesTests :: TestTree
parseParanthesTests =
  testGroup
    "Parantheses"
    [ testCase "Parantheses" $ parseString ("(1)") @?= Right (Number 1)
    , testCase "Parantheses whitespace" $
      parseString ("(   1     )") @?= Right (Number 1)
    ]

parseExprs :: TestTree
parseExprs =
  testGroup
    "parseExprs"
    [ testCase "parseExprs numbers" $
      parseString ("[1,2,3]") @?= Right (Array [Number 1, Number
       ↪  2, Number 3])
    , testCase "parseExprs" $
      parseString ("['a','b','c']") @?=
      Right (Array [String "a", String "b", String "c"])
    , testCase "parseExprs ident" $
      parseString ("a (1,2,3)") @?=
      Right (Call "a" [Number 1, Number 2, Number 3])
    ]

parseComma :: TestTree
parseComma =
  testGroup
    "Comma"
    [ testCase "Parse Comma" $
      parseString ("1,2") @?= Right (Comma (Number 1) (Number 2))
    , testCase "Parse nested commas" $
      parseString ("1,(1,(3,4))") @?=
      Right (Comma (Number 1) (Comma (Number 1) (Comma (Number 3)
       ↪  (Number 4))))
    , testCase "many commas" $
      parseString ("1,2,3,'a','b'") @?=
      Right
        (Comma
           (Number 1)
           (Comma
              (Number 2)
              (Comma (Number 3) (Comma (String "a") (String
               ↪  "b")))))
    ]

parseExprTests :: TestTree
parseExprTests =
  testGroup
    "parseExpr"
    [ testCase "Additon" $
      parseString ("1+1") @?= Right (Call "+" [Number 1, Number
       ↪  1])
    , testCase "Subtraction" $
      parseString ("1-1") @?= Right (Call "-" [Number 1, Number
       ↪  1])
    , testCase "Mul" $
      parseString ("1*1") @?= Right (Call "*" [Number 1, Number
       ↪  1])
    , testCase "Mod" $
      parseString ("1%1") @?= Right (Call "%" [Number 1, Number
       ↪  1])
    , testCase "Smaller Then" $
      parseString ("1<1") @?= Right (Call "<" [Number 1, Number
       ↪  1])
    , testCase "Equals" $
      parseString ("1===1") @?= Right (Call "===" [Number 1,
       ↪  Number 1])
    ]

parseArrayCompr :: TestTree
parseArrayCompr =
  testGroup
    "Array Compr"
    [ testCase "for" $
      parseString ("[for (x of 2) 3]") @?=
      Right (Compr (ACFor "x" (Number 2) (ACBody (Number 3))))
    , testCase "nested for" $
      parseString ("[for (x of 2) for (x of 3) 3]") @?=
      Right
        (Compr (ACFor "x" (Number 2) (ACFor "x" (Number 3)
         ↪  (ACBody (Number 3)))))
    , testCase "nested if" $
      parseString ("[for (x of 2) if(1) 2]") @?=
      Right (Compr (ACFor "x" (Number 2) (ACIf (Number 1) (ACBody
       ↪  (Number 2)))))
    , testCase "mixed for/if" $
      parseString ("[for (x of 2) if(1) for (y of 2) if(false)
       ↪  for(z of 5) 2]") @?=
      Right
        (Compr
           (ACFor
              "x"
              (Number 2)
              (ACIf
                 (Number 1)
                 (ACFor
                    "y"
                    (Number 2)
                    (ACIf FalseConst (ACFor "z" (Number 5)
                     ↪  (ACBody (Number 2))))))))
    ]

constantTests :: TestTree
constantTests =
  testGroup
    "constants tests"
    [ testCase "Number" $ parseString ("2") @?= Right (Number 2)
    , testCase "String" $ parseString ("'abc'") @?= Right (String
    ↪  "abc")
    , testCase "true" $ parseString ("true") @?= Right
    ↪  (TrueConst)
    , testCase "false" $ parseString ("false") @?= Right
    ↪  (FalseConst)
    , testCase "Undefined" $ parseString ("undefined") @?= Right
    ↪  (Undefined)
    , testCase "Ident" $ parseString ("sdsd") @?= Right (Var
    ↪  "sdsd")
    ]

parseSimpleExprTests :: TestTree
parseSimpleExprTests =
  testGroup
    "Simple expr tests"
    [ testCase "equal" $
      parseString ("a===b===c") @?=
      Right (Call "===" [Call "===" [Var "a", Var "b"], Var "c"])
    , testCase "assign" $
      parseString ("a=b=undefined") @?=
      Right (Assign "a" (Assign "b" Undefined))
    , testCase "smallerThen" $
      parseString ("2<3<4") @?=
      Right (Call "<" [Call "<" [Number 2, Number 3], Number 4])
    , testCase "whitespace" $
      parseString ("12    \v \t\t     \n") @?= Right (Number 12)
    , testCase "comment" $
      parseString ("1 //comment 11212121212\n,2") @?=
      Right (Comma (Number 1) (Number 2))
    , testCase "comment at start" $
      parseString ("//comment \n 2    ") @?= Right (Number 2)
    ]

parseComplexExprTests :: TestTree
parseComplexExprTests =
  testGroup
```

```
"Complex expr tests"
[ testCase "scope.js" $
  parseString ("x = 42, y = [for (x of 'abc') x],[x, y]") @?=
  Right
    (Comma
       (Assign "x" (Number 42))
       (Comma
          (Assign "y" (Compr (ACFor "x" (String "abc")
          ↪   (ACBody (Var "x")))))
          (Array [Var "x", Var "y"])))
, testCase "correct precedence add" $
  parseString ("[1,2,3,4] + [1,2,3]") @?=
  Right
    (Call
       "+"
       [ Array [Number 1, Number 2, Number 3, Number 4]
       , Array [Number 1, Number 2, Number 3]
       ])
, testCase "precedences" $
  parseString ("1+2*4-3%4") @?=
  Right
    (Call
       "-"
       [ Call "+" [Number 1, Call "*" [Number 2, Number 4]]
       , Call "%" [Number 3, Number 4]
       ])
, testCase "arrayCompr complex" $
  parseString
    ("[for (a of 4) 1] * [for (a of abc) if (true) if (false)
    ↪   2*3]") @?=
  Right
    (Call
       "*"
       [ Compr (ACFor "a" (Number 4) (ACBody (Number 1)))
       , Compr
           (ACFor
              "a"
              (Var "abc")
              (ACIf
                 TrueConst
                 (ACIf FalseConst (ACBody (Call "*" [Number
                 ↪   2, Number 3]))))))
       ])
]
```

```

parseErrorTest :: TestTree
parseErrorTest =
  testGroup
    "Parse Fail"
    [ testCase "let parser fail" $
      show (parseString ("")) @?=
      "Left (ParseError \"\\\"ERROR\\\" (line 1, column
       ↪  1):\\nunexpected end of input\\nexpecting white space,
       ↪  \\\"//\\\", \\\"[\\\", letter, \\\"(\\\", digit,
       ↪  \\\"-\\\", \\\"'\\\", \\\"true\\\", \\\"false\\\" or
       ↪  \\\"undefined\\\"\")"
    ]

predefinedTests :: TestTree
predefinedTests =
  testGroup
    "predefined tests"
    [ testCase "tiny" $
      parseString "2+3" @?= Right (Call "+" [Number 2, Number 3])
    , testCase "intro" $ do
        act <- parseFile "examples/intro.js"
        exp <- fmap read $ readFile "examples/intro-ast.txt"
        act @?= Right exp
    ]
```

```
module ParserUtils where

import SubsAst
import SubsParser
import Text.Parsec.Char
import Text.Parsec.Combinator
import Text.Parsec.Prim
import Text.Parsec.String
import Text.Parsec.Error

-- for testing parseNumber
numberParser :: String -> Either ParseError Expr
numberParser s =
  parse
    (do res <- parseNumber
        eof
        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
```

```
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
```