Haskell intro

Assignment 2

Kai Arne S. Myklebust, Silvan Adrian

Handed in: September 25, 2018

Contents

1 Design/Implementation

1.1 Choice of Parser combinator Library

We decided to use **Parsec** because of the better error handling possabilities compared to **ReadP**.

1.2 Whitespace

1.3

2 Code Assessment

2.1 Tests

2.2 Test Coverage

2.3 OnlineTA

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
import           Text.Parsec.String
--import           Data.Char

parseString :: String -> Either ParseError Expr
parseString s = parse (do
                        res <- parseLeadingWhitespace parseExpr
                        eof
                        return res) "ERROR" s

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"
```

```haskell

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

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 ]

```

```haskell
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


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

parseIdent :: Parser Expr
parseIdent = do
                fc <- letter
                rest <- many (digit <|> letter <|> char '_')
                return (Var (fc:rest::String))

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
```

```haskell
                _ <- 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 [ parseACBody, parseArrayFor, parseACIf
  ↪  ]


parseACBody :: Parser ArrayCompr
parseACBody = do
                expr <- parseExpr'
```

```
158                 return (ACBody expr)
159
160 parseACIf :: Parser ArrayCompr
161 parseACIf = do
162                 _ <- string "if"
163                 _ <- char '('
164                 expr1 <- parseExpr'
165                 _ <- char ')'
166                 compr <- parseArrayCompr
167                 return (ACIf expr1 compr)
168
169
170 parseArray :: Parser Expr
171 parseArray = do
172                 _ <- parseWhitespace(char '[')
173                 exprs <- parseExprs
174                 _ <- parseWhitespace(char ']')
175                 return (Array exprs)
176
177 -- isLegalChar :: Char -> Bool
178 -- isLegalChar c | ord c >= 32 && ord c <= 126 = True
179 --               | otherwise = False
180
181 parseStr :: Parser Expr
182 parseStr = do
183                 _ <- parseWhitespace(char '\'')
184                 res <- parseWhitespace(many alphaNum)
185                 --res <- many (satisfy isLegalChar)
186                 _ <- parseWhitespace(char '\'')
187                 return (String res)
188
189 parseTrue :: Parser Expr
190 parseTrue = do
191                 _ <- string "true"
192                 return TrueConst
193
194 parseFalse :: Parser Expr
195 parseFalse = do
196                 _ <- string "false"
197                 return FalseConst
198
199 parseUndefined :: Parser Expr
200 parseUndefined = do
201                     _ <- string "undefined"
```

```
202                     return Undefined
203
204
205 parseExpr' :: Parser Expr
206 parseExpr' = parseAdditon `chainl1` parseCompare
207
208 parseCompare :: Parser (Expr -> Expr -> Expr)
209 parseCompare = (do
210               _ <- parseWhitespace(string "<")
211               return (\x y -> Call "<" [x, y]))
212               <|> (do
213                       _ <- parseWhitespace(string "===")
214                       return (\x y -> Call "===" [x, y]))
215
216 parseAdditon :: Parser Expr
217 parseAdditon = do
218                   prod <- parseProd
219                   parseAdditon' prod
220
221 parseAdditon' :: Expr -> Parser Expr
222 parseAdditon' input = (do
223                         addOp <- parseWhitespace(char '+' <|>
    ↪  char '-')
224                         cons <- parseProd
225                         parseAdditon' $ Call [addOp] [input,
    ↪  cons])
226                         <|> return input
227
228 parseProd :: Parser Expr
229 parseProd = do
230                 cons <- parseCons
231                 parseProd' cons
232
233 parseProd' :: Expr -> Parser Expr
234 parseProd' input = (do
235                       prodOp <- parseWhitespace(char '*' <|>
    ↪  char '%')
236                       cons <- parseCons
237                       parseProd' $ Call [prodOp] [input, cons])
238                       <|> 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
import Text.ParserCombinators.Parsec.Error

main = defaultMain tests

tests :: TestTree
tests =
  testGroup
    "Tests"
    -- predefinedTests
    [ constantTests
    , parseNumberTests
    , parseStringTests
    , parseFalseTests
    , parseTrueTests
    , parseUndefinedTests
    , parseAssignTests
    , parseCallTests
    , parseIdentTests
    , parseArrayTests
    , parseStartArrayTests
    ]

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)
```

```
40      , testCase "Number 8 long neg" $
41        numberParser ("-12345678") @?= Right (Number (-12345678))
42      , testCase "Number too long pos" $
43        show (numberParser ("123456789")) @?=
44        "Left \"ERROR\" (line 1, column 10):\nunexpected end of
    ↪  input\nexpecting digit\nNumber too long"
45      , testCase "Number too long neg" $
46        show (numberParser ("-123456789")) @?=
47        "Left \"ERROR\" (line 1, column 11):\nunexpected end of
    ↪  input\nexpecting digit\nNumber too long"
48      ]
49
50  parseStringTests :: TestTree
51  parseStringTests =
52    testGroup
53      "parse string"
54      [ testCase "String" $ stringParser ("'abc'") @?= Right
    ↪  (String "abc")
55      , testCase "String alphaNum" $
56        stringParser ("'abc123'") @?= Right (String "abc123")
57      , testCase "String allowed special chars" $
58        stringParser ("'abc\n\t") @?= Right (String "abc")
59      , testCase "String  not allowed special char" $
60        stringParser ("'\a'") @?= Right (String "Error")
61      , testCase "String whitespaced" $
62        stringParser ("'asdas asdasd'") @?= Right (String "asdas
    ↪  asdasd")
63      , testCase "String newline" $
64        stringParser ("'foo\\\nbar'") @?= Right (String "foobar")
65      ]
66
67  parseFalseTests :: TestTree
68  parseFalseTests =
69    testGroup
70      "parse false"
71      [ testCase "False" $ falseParser ("false") @?= Right
    ↪  (FalseConst)
72      , testCase "False fail" $
73        show (falseParser ("true")) @?=
74        "Left \"ERROR\" (line 1, column 1):\nunexpected
    ↪  \"t\"\nexpecting \"false\""
75      ]
76
77  parseTrueTests :: TestTree
```

```
78  parseTrueTests =
79    testGroup
80      "parse true"
81      [ testCase "True" $ trueParser ("true") @?= Right (TrueConst)
82      , testCase "True fail" $
83        show (trueParser ("false")) @?=
84        "Left \"ERROR\" (line 1, column 1):\nunexpected
    ↪  \"f\"\nexpecting \"true\""
85      ]
86
87  parseUndefinedTests :: TestTree
88  parseUndefinedTests =
89    testGroup
90      "Undefined"
91      [ testCase "Undefined" $ undefinedParser ("undefined") @?=
    ↪  Right (Undefined)
92      , testCase "Undefined fail" $
93        show (undefinedParser ("defined")) @?=
94        "Left \"ERROR\" (line 1, column 1):\nunexpected
    ↪  \"d\"\nexpecting \"undefined\""
95      ]
96
97  parseAssignTests :: TestTree
98  parseAssignTests = testGroup "Assign"
99    [
100     testCase "Assign" $ assignParser("x=3") @?= Right (Assign "x"
    ↪  (Number 3)),
101     testCase "Assign whitespace/special char" $ assignParser("x =
    ↪  \n 3") @?= Right (Assign "x" (Number 3)),
102     testCase "Assign underline" $ assignParser("x_x=0") @?= Right
    ↪  (Assign "x_x" (Number 0))
103   ]
104
105 parseCallTests :: TestTree
106 parseCallTests = testGroup "Call"
107   [
108     testCase "Call" $ callParser("x(12)") @?= Right (Call "x"
    ↪  [Number 12]),
109     testCase "Call whitespace" $ callParser("x ( 12 ) ") @?=
    ↪  Right (Call "x" [Number 12])
110   ]
111
112 parseIdentTests :: TestTree
113 parseIdentTests = testGroup "Ident"
```

```
114     [
115       testCase "Ident" $ identParser("x_x") @?= Right (Var "x_x"),
116       testCase "Ident keyword" $ identParser("falsee") @?= Right
      ↪  (Var "falsee"),
117       testCase "Ident whitespace" $ show(identParser("x_x    "))
      ↪  @?= "Left \"ERROR\" (line 1, column 4):\nunexpected '
      ↪  '\nexpecting digit, letter, \"_\" or end of input"
118     ]
119
120   parseArrayTests :: TestTree
121   parseArrayTests = testGroup "Array"
122     [
123       testCase "Array" $ parseString("[1,2]") @?= Right (Array
      ↪  [Number 1,Number 2]),
124       testCase "Array whitespace" $ parseString("[ 1,  'sds']   ")
      ↪  @?= Right (Array [Number 1,String "sds"])
125     ]
126
127
128   parseStartArrayTests :: TestTree
129   parseStartArrayTests = testGroup "Array Compr"
130     [
131       testCase "Array for" $ parseString("[for (x of 2) 2]") @?=
      ↪  Right (Compr (ACFor "x" (Number 2) (ACBody (Number 2)))),
132       testCase "Array whitespace" $ parseString("[ 1,  'sds']   ")
      ↪  @?= Right (Array [Number 1,String "sds"])
133     ]
134
135   constantTests :: TestTree
136   constantTests =
137     testGroup
138       "constants tests"
139       [ testCase "Number" $ parseString ("2") @?= Right (Number 2)
140       , testCase "String" $ parseString ("'abc'") @?= Right (String
      ↪  "abc")
141       , testCase "true" $ parseString ("true") @?= Right
      ↪  (TrueConst)
142       , testCase "false" $ parseString ("false") @?= Right
      ↪  (FalseConst)
143       , testCase "Undefined" $ parseString ("undefined") @?= Right
      ↪  (Undefined)
144       , testCase "Ident" $ parseString ("sdsd") @?= Right (Var
      ↪  "sdsd")
145       ]
```

```
146
147 predefinedTests :: TestTree
148 predefinedTests =
149   testGroup
150     "predefined tests"
151     [ testCase "tiny" $
152       parseString "2+3" @?= Right (Call "+" [Number 2, Number 3])
153     , testCase "intro" $ do
154         act <- parseFile "examples/intro.js"
155         exp <- fmap read $ readFile "examples/intro-ast.txt"
156         act @?= Right exp
157     ]
```