

Name: Chakrya Ros

## Project Report : Parser Lettuce

I decided to implement Parse Lettuce for this project. Firstly, I am doing research about parsing. I read some articles that are useful for me to start my project. I found one article on github that explain how to use parsing combinator library. And I read the book in library, the Programming in Scala. It's also helpful for me to start my project.

After I read those documents, and I started writing the definition and inference rule.

ParserLettuce ::= identifier (Parser[Expr])

- | boolParse (Parser[Expr])
- | PositiveParser (Parser[Expr])
- | NegativeParser (Parser[Expr])
- | symbol (Parser[String])
- | Logic\_Parser (Parser[String])
- | If\_Parser (Parser[String])
- | then\_Parser (Parser[String])
- | else\_Parser (Parser[String])
- | Let\_Parser (Parser[String])
- | in\_Parser (Parser[String])
- | LetRec\_Parser(Parser[String])
- | expr( identifier, boolParser, PositiveParser, NegativeParser, eval\_let,  
eval\_FunDef, eval\_FunCall, eval\_LetRec, eval\_logic ,eval\_equals)
- | eval\_Arth( expr, symbol, expr)
- | eval\_ifThenElse(If\_Parser, expr, then\_Parser, expr, else\_Parser expr)
- | eval\_Let( Let\_Parser, expr, equals, expr, in\_Parser, expr)
- | eval\_FunDef(String, "(" , String, ")", expr)
- | eval\_FunCall("(" , expr, "[", expr "]", ")")
- | eval\_LetRec( LetRec, str, equals, str, "(" , expr, ")" expr, in\_Parser, expr)
- | apply(String)

These are inference rules:

$$\frac{i \in [a-zA-Z] , i \Downarrow x}{\text{identifier}(i) \Downarrow \text{Ident}(x)} \quad (\text{identifier})$$

$$\frac{e \in [\text{"true"} | \text{"false"} | \text{"false"} | \text{"true"}] , e \Downarrow p}{\text{boolParser}(e) \Downarrow \text{Bin}(p)} \quad (\text{boolParser})$$

$$\frac{i \in [0 | 1-9] \Downarrow \text{num} , \text{Int-to-num}(i)}{\text{PositiveParser}(i) \Downarrow \text{Const}(\text{Positive}(i))} \quad (\text{PositiveParser})$$

$$\frac{i \in [0 | 1-9] \Downarrow \text{num} , \text{Int-to-num}(i)}{\text{NegativeParser}(-i) \Downarrow \text{Const}(\text{Negative}(i))} \quad (\text{NegativeParser})$$

$$\frac{i \in [\text{"+"} | \text{"-"} | \text{"*"} | \text{"^"}] , i \Downarrow s}{\text{symbol}(e) \Downarrow s} \quad (\text{symbol})$$

$$\frac{e_1 \Downarrow \text{const}(e_1) , e_2 \Downarrow \text{const}(e_2) , s \in \text{Symbol}}{\text{eval-Arith}(e, s, e) \Downarrow f(e_1, e_2)} \quad (\text{eval-Arith})$$

$$\text{where } f_+ e_1, e_2 = \text{Plus}(e_1, e_2)$$

$$f_- e_1, e_2 = \text{Minus}(e_1, e_2)$$

$$f_{\times} e_1, e_2 = \text{Mult}(e_1, e_2)$$

$$f_{\wedge} e_1, e_2 = \text{Pow}(e_1, e_2)$$

$$\frac{\text{if} \Downarrow \text{"if"} , x \Downarrow \text{Expr} , \text{then} \Downarrow \text{"Then"} , e_1 \Downarrow \text{Expr} , \text{else} \Downarrow \text{"Else"} , e_2 \Downarrow \text{Expr}}{\text{eval-IfThenElse}(\text{if}, x, \text{then}, e_1, \text{else}, e_2) \Downarrow \text{IfThenElse}(x, e_1, e_2)} \quad (\text{eval-IfThenElse})$$

$$\frac{x \Downarrow \text{Expr} , y \Downarrow \text{Expr} , e_q \Downarrow \text{"="}}{\text{eval-equals}(x, e_q, y) \Downarrow \text{Eq}(x, y)} \quad (\text{Eval-equals})$$

$$\frac{x \Downarrow \text{Expr} , y \Downarrow \text{Expr} , \text{logic} \Downarrow \text{"and"} | \text{"and"} | \text{"or"} | \text{"or"} , f \in \{\text{and}, \text{or}\}}{\text{eval-logic}(x, \text{logic}, y) \Downarrow f(x, y)} \quad (\text{Eval-logic}) \quad \text{where } f_{\text{and}}(x, y) = \text{And}(x, y) \\ \text{for } f_{\text{or}}(x, y) = \text{Or}(x, y)$$

$$\frac{e_1 \Downarrow \text{let}, s \Downarrow \text{string}, eq \Downarrow \text{equals}, e_2 \Downarrow \text{Expr}, in \Downarrow \text{"in"}, e_4 \Downarrow \text{Expr}}{\text{eval\_let}(e_1, x, eq, Fun, in, body) \Downarrow \text{Let}(x, Fun, body)} \quad (\text{eval\_let})$$

$$\frac{str \Downarrow \text{"Function"}, op \Downarrow \text{"["}, id \Downarrow [a-zA-z], cp \Downarrow \text{"}"}, body \Downarrow \text{Expr}}{\text{eval\_FunDef}(str, op, id, cp, body) \Downarrow \text{FunDef}(id, body)} \quad (\text{Eval\_FunDef})$$

$$\frac{e_1 \Downarrow \text{Expr}, e_2 \Downarrow \text{Expr}}{\text{eval\_FunCall}(e_1, e_2) \Downarrow \text{FunCall}(\text{Expr}, \text{Expr})} \quad (\text{eval\_FunCall})$$

$$\frac{\begin{array}{l} f \in \{ \text{boolParser}, \text{positiveParser}, \text{negativeParser}, \text{idParser}, \text{eval\_Arith}, \text{eval\_IfThenElse}, \\ \text{eval\_let}, \text{eval\_FunDef}, \text{eval\_LetRec}, \text{eval\_equals}, \text{Eval\_logrc} \} \end{array}}{\text{expr}(e) \Downarrow f} \quad (\text{Expr})$$

$$\frac{L \Downarrow \text{LetRec}, id \Downarrow id, e \Downarrow \text{equals}, op \Downarrow \text{"["}, x \Downarrow [a-zA-z], cp \Downarrow \text{"}"}, FunDef \Downarrow \text{Expr}, in\_parser \Downarrow \text{"in"}, body \Downarrow \text{Expr}}{\text{eval\_LetRec}(L, id, e, f, op, x, cp, FunDef, in\_parser, body) \Downarrow \text{LetRec}(id, x, FunDef, body)}$$

I have learned some symbols like

$p1 \sim p2$  mean sequencing: must match  $p1$  followed by  $p2$ .

$p1 \mid p2$  mean alternation: must match either  $p1$  or  $p2$ , with preference given to  $p1$ .

$p1.?$  mean optionality: may match  $p1$  or not

$p1.*$  mean repetition: matches any number of repetitions of  $p1$

These help me a lot for parsing lettuce. And then I started to write my code, I had struggled with "Import scala.util.parsing.combinator.\_". It kept me a error. I don't understand why, so I had to ask TA (Benno) for help about this before continuing. He could not find out the problem yet. So I decided to download IntelliJ IDEA CA for started writing the code. I wrote eval\_Arith function that take two expr and symbol (+, -, \*, ^), and then I tested (5+2), it's passed the test.

```
417 // else Succ(number_to_nat(i-1))
418
419
420 //Check if positive or negative
421 def Int_to_Const(args: Int): Const = {
422   if (args > 0)
423     Const(Positive(number_to_nat(args)))
424   else
425     Const(Negative(number_to_nat(-args)))
426 }
427
428 def Int_to_Ident(args: String): Ident = Ident(args)
429
430
431
432
433 // p1 ~ p2 // sequencing: must match p1 followed by p2
434 // p1 | p2 // alternation: must match either p1 or p2, with preference given to p1
435 // p1? // optionality: may match p1 or not
436 // p1* // repetition: matches any number of repetitions of p1
437 def identifier: Parser[Ident] = "[a-zA-Z_][a-zA-Z0-9_]*".r ^^ {(x => Ident(x)) }
438
439 def number: Parser[Const] = "[0-9]+".r ^^ { x => Int_to_Const(x.toInt) }
440
441 def symbol: Parser[String] = "+" | "-" | "*"
442
443 def equals : Parser[Any] = "="
444
445 def let : Parser[String] = "Let"
446
447 def in : Parser[String] = "in"
448
449
450 def expr : Parser[Expr] = identifier | number | eval_Arth
451
452
453 def eval_Arth = {
454   expr ~ symbol ~ expr ^^
455   {
456     case (e1 ~ "+" ~ e2) => Plus(e1, e2)
457     case (e1 ~ "-" ~ e2) => Minus(e1, e2)
458     case (e1 ~ "*" ~ e2) => Mult(e1, e2)
459   }
460 }
461
462 TokenFromInput() eval_Arth()
```

However, I tested  $(5+2+3)$ , it's failed. Then I had to do another research for this problem. I could not figure out this problem. I visited Benno's office Hour for help. Benno found the way to make Jupyter Notebook work. So I had to move all my code from IntelliJ IDEA to Jupyter Notebook. It's much better to use Jupyter Notebook because I can run my code and see output what I want. I asked Benno why my code didn't work for testing several operators  $(5+2*2)$  together. He found out that we need to use parentheses for every operator. For example,  $((3+2)*4)$ ,  $((8-2)+3)-1$

Everything has been working fine for arithmetic. I started to write logic "and" and "or" and "eq" function. Those was so easy to write. However, When I started to do let, FunDef, FunCall and LetRec, it took me more than five hours to get it done. I went to ask Benno for help to write LetRec. Finally, I got all of them work with Check\_Assert function that I wrote to test all the function definition.

```

def Check_Assert(x : Expr, expected : Value): Unit = {
    assert(eval(EmptyEnv, x) == expected)
    println("Pass Test!!!")
}

```

After that, I researched how to do REPL for extra credit. It's not bad for me. I just create scanner variable, and import "new java.util.Scanner(System.in)" to read user input. I wrote function readInput that take no argument. I use while loop for repeating the user input, and Boolean for stopping the while loop. When I get user input, I parsed input to ParserLettuce class that I wrote and evaluate the output.

```

1  def readInput {
2
3      var bool_ = true
4      var scanner = new java.util.Scanner(System.in)
5      println("Welcome to Scala!!!!")
6      println("Please enter your input or (quit) to quit")
7      while (bool_ == true) {
8          var userInput = scanner.nextLine()
9          while(userInput == "")|
10         {
11             println("Your input was empty. Please enter your input again.")
12             println()
13             userInput = scanner.nextLine()
14             if(userInput == "quit")
15             {
16                 bool_ = false
17             }
18         }
19         if(userInput == "quit")
20         {
21             bool_ = false
22         }
23         else {
24             var parsed = ParserLettuce(userInput)
25             var eval_ = eval(EmptyEnv, parsed)
26             println("Your input is " + userInput)
27             println()
28             println("Evaluated > " + eval_)
29         }
30     }
31 }
32

```

defined function readInput

Overall, I have spent more than 10 hours to do this project. I feel I had learned a lot from this project. It's not really difficult as I expected.

## Reference:

1. Programming in Scala, Third Edition by Bill Venners, Lex Spoon, Martin Odersky
2. <https://github.com/enear/parser-combinators-tutorial/blob/master/src/main/scala/co/enear/parsercombinators/lexer/WorkflowLexer.scala>
3. <https://www.cs.helsinki.fi/u/wikla/OTS/Sisalto/examples/html/ch31.html>
4. <https://www.scala-lang.org/api/2.12.2/scala-parser-combinators/scala/util/parsing/combinator/Parsers.html#Input=scala.util.parsing.input.Reader%5BParsers.this.Elem%5D>