PPL Project Report

For our project, we chose to implement a specific kind of LISP known as Scheme. The goal was to create this implementation in Scala, the language we have been working closely with throughout our time in Principles of Programming Languages. To begin this project, we first researched how Scheme worked, what the syntax was, and how the semantics worked. From there we created a list of expressions, for our implementation of Scheme. We drew inspiration from the functions we implemented in Lettuce and started with the basics:

```
Value  ::= NumVal(Integer)
         | BinVal(Bool)
         | ClosureSimple(List[String],Expr, String)
         | Error
         | Plus_
         | Minus_
         | Mult_
         | Pow_
         | Neg_
         | Eg_
         | And_
         | Or_|

Expr  ::= Const(Integer)
        | Bin(Bool)
        | Ident(String)
        | Pow(Expr, Expr)
        | Plus(Expr, Expr)
        | Minus(Expr, Expr)
        | Mult(Expr, Expr)
        | And(Expr, Expr)
        | Or(Expr, Expr)
        | Neg(Expr)
        | Eq(Expr, Expr)
        | IfThenElse(Expr, Expr, Expr)
        | SymbolExpr(String)
        | Define(String, List[Expr], Expr)
        | Pair(Expr, Expr)
```

In creating the lists and writing the rules, we realized that many of these functions have the same syntax, the same implementation, and the same functional processing as the ones we wrote for the Lettuce interpreter. However, not all were the same. The Scheme interpreter does not include FunCall, FunDef, Let, and LetRec. The functionalities we did not include in Scheme are handled now by three new definition types: SymbolExpr, Define, and Pair.

We also knew we needed to create a Parse function to handle the syntax of Scheme. Now, the syntax of Scheme contains either an operator such as +, -, * and values to be manipulated. An example of this is:

(+ 2 3)

In this example, it is saying to add 2 and 3 together, however it is wrapped in parenthesis and there are spaces in between each of the items in the parenthesis. Other cases must be handled as well such as:

(define f (x y) (+ 2 3))

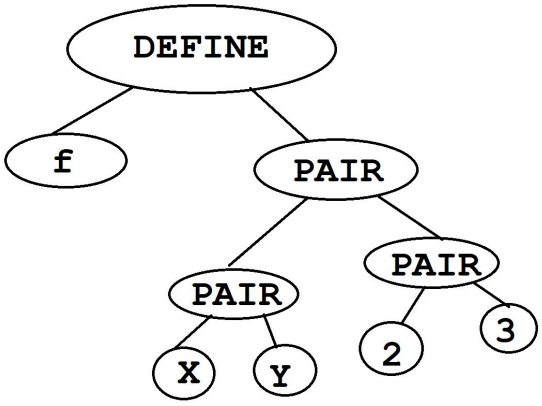
In this case, we have the string “define”, which is telling us that something special needs to happen to this function, thus we will handle the parsing of this function differently than the previous function.

To handle the syntax Scheme, our Parse function must separate the parentheses and every part of the define statement. Our group struggled a lot with this specific aspect, trying to figure out how to use Scala best in this situation. When Scheme gives our Scala interpreter the function, it comes in a string. From there we have to break up that string into a list, breaking apart every section and keeping the parentheses as well, yet getting rid of the spaces. We ended up having to look up a multitude of ways online to try and figure out how to parse each individual element.

We solved the first initial step of parsing by using built-in Scala functions. We used “.split”, and split apart letters from the parentheses using Regular Expressions (RegEx). Attached

to it we used the ".filter" to get rid of all spaces in our string, then used ".toList" to put our strings into a list. From there, we had created a list, and using the above "define" function, our list would be: ["define", "f", "(", "x", "y", ")", "2", "3", ")", ")"].

Now that the Scheme list had been parsed into a list, our Parse function would need to handle what to do next. In Scheme, everything comes in pairs. From this point, we would need to put together this list into an Abstract Syntax Tree. To do this we will have the following image below:



Our Parse function must contain a way to put our resulting list into pairs. From there, we checked to see if our list contains a "define". If it does, we check to see where and how many pairs of parentheses there are in the list. This will determine how we separate the pairs, to pass to our Eval. An example of this would be if we are given (define f (+ 5 (- 2 3)) using our parse and then putting back together with Pair would result in, Pair(+, Pair(5, Pair(-, Pair(2 3)))). This proved to be a difficult task, making a list of Pairs that we would send to Eval to be evaluated. However, if the list passed does not contain "define" we would check for the symbol that is in our list and send to Eval to be evaluated. Parse has so far shown to be the most difficult task for us. A majority of our time we spent collectively and apart, we were trying to create and implement Parse.

From there, we started to implement our functions in Eval. As we stated above, we were able to reuse many of our functions from Lettuce. However, we needed to make three more, SymbolExpr, Pair and Define.

SymbolExpr is a type of Expression that takes a character, and returns which function to use. If the character passed to SymbolExpr is '+', then our program will know that the function needs to add the following pair together. It returns in this case "Plus_" which will be used later. If the character passed to SymbolExpr is '-', then our program will know that the function will need to subtract the following pair together and return "Minus_". However, that computation does not happen in SymbolExpr and is instead handled in Pair. Below is how we implemented SymbolExpr:

```
case SymbolExpr(x) => x match {
    case "+" => Plus_
    case "-" => Minus_
    case "*" => Mult_
    case "expt" => Pow_
    case "not" => Neg_
    case "eq" => Eq_
    case "and" => And_
    case "or" => Or_
    case "if" => If_
}
```

Pair is a type of Expression that takes a pair of expressions, for example, Pair(Plus_ Pair(2 3)) and evaluates it. In this case, it will match on "Plus_" and evaluate "2 3" using our Plus definition that was reused from our Lettuce interpreters and will output '5'. Below is how we implemented Pair:

```
case Pair(x,y) => (eval(env,x),y) match {
    case (Plus_,Pair(e1,e2)) => eval(env,Plus(e1,e2))
    case (Minus_,Pair(e1,e2)) => eval(env,Minus(e1,e2))
    case (Mult_,Pair(e1,e2)) => eval(env,Mult(e1,e2))
    case (Pow_,Pair(e1,e2)) => eval(env,Pow(e1,e2))
    case (Neg_,e1) => eval(env,Neg(e1))
    case (Eq_,Pair(e1,e2)) => eval(env,Eq(e1,e2))
    case (And_,Pair(e1,e2)) => eval(env,And(e1,e2))
    case (Or_, Pair(e1,e2)) => eval(env,Or(e1,e2))
    case (If_, Pair(e1,Pair(e2,e3))) => eval(env,IfThenElse(e1,e2,e3))
}
```

Define is a type of Expression that is specifically tied into the Scheme syntax, as seen above with (define f (x y) (+ 2 3)). For this function in our Eval, it takes in a string, a list of expressions that we call our arguments, and an expression that is our body. This function is to handle what the Parse function sends Eval when the Parse function finds a “define”. If there is no list of arguments, then only the body will be evaluated. However, if there is a list of arguments, then our define would evaluate to a Closure we named ClosureSimple. ClosureSimple functions similarly to the Closure defined in Lettuce, however instead of taking in just a string that would be the string name, ClosureSimple takes in a list of arguments, and then the body and an environment. Below is how we implemented Define:

```
case Define(id, args, body) => args match {
    case Empty => {
        val new_env = Extend(id,eval(env,body),env)
        eval(new_env,body)
    }
    case _ => {
        val new_env = Extend(id,ClosureSimple(args,body,env),env)
        ClosureSimple(args,body,env)
    }
}
```

Overall, this project proved to be more difficult than we expected. The Parse function took the most amount of our time to understand and to implement. However, it is uncanny how similar, the Lettuce Interpreter and the Scheme Interpreters ended up being with respect to how the arithmetic functions operated. Yet they are different with how functions are handled, specifically that our Scheme Interpreter had SymbolExpr, Pair and Define instead of FunDef, FunCall and Let in Lettuce.

Outline

Defining Syntax/Semantics: *Group*

Parsing: *Group / Cassie*

Eval: *Group/Cassie*

Test Cases: *Cassie*

Report: *Mattie / Muntaha*

Sources

These are the sources we used in order to understand and implement our project. What we used these sources for is written above the links.

1. Mutable Lists in Scala

https://alvinalexander.com/scala/how-to-create-mutable-list-in-scala-listbuffer-cookbook

2. Example Interpreter

https://martintrojer.github.io/scala/2013/06/06/scheme-in-scala

3. Understanding Scheme

http://www.r6rs.org/final/r6rs.pdf

4. Scheme Guidebook

https://cs.nyu.edu/courses/summer06/G22.2110-001/web/lectures/lec01-scheme.txt

5. Understanding Conditionals in Scheme

https://en.wikibooks.org/wiki/Scheme_Programming/Conditionals

6. Parsing with Regex

https://stackoverflow.com/questions/19551936/parsing-a-list-with-regular-expressions

7. List Traversals

https://alvinalexander.com/scala/scala-for-loops-foreach-how-to-translated-by-compiler

8. Introduction Video to Scheme Syntax

https://www.youtube.com/watch?v=6k78c8EctXI

9. Scala Mutable List

https://stackoverflow.com/questions/52425954/how-to-append-elements-to-list-in-a-for-loop-scala