

## CS 320 Midterm essay

20180109 김수빈

### 1. “Growing a language”

#### 1-1 Summary

The speaker starts the speech by defining ‘man’ and ‘woman’. Defining words with more than one syllable that is used in his speech is one of the unique aspects of this talk. This way of speaking is quite similar to building a programming language, in a way that it starts with the smallest elements. It also emphasizes the importance of defining primitive elements in a language. According to the speaker, there are two kind of programming languages: a big one, and a small one. Most programs tend to start small, and grow later on as users modify and patch warts in the language. If a language starts big in the first place, it is hard to learn and would easily be shunned aside by smaller, easier, quicker languages. A language too small is difficult to use as well. Because of the lack of tools, the user may not be able to express what is needed. This is the same with “a right language”. Trying to make a language that is organized and logically right in every aspect from the start is ideal, but the speaker doubts its possibility. Instead, he encourages the growth of a language, and to grow right. He gives a metaphor about cathedral and bazaar based on “The Cathedral and the Bazaar” by Eric Raymond. If the builder gives a master plan, it is the cathedral. If people can change the environment they are using meeting their needs in real time, it is a bazaar. What is ideal for the speaker is this: “Instead of a designing a thing, you need to design a way of doing.”. He talks about designing a way of designing a programming language. To grow a language, one must choose what must be implemented. Is the term ‘complex numbers’ worth implementing? Is this way of calculating useful and worth the effort of implementing? In this procedure of questioning, language can grow towards the purpose that users need. Designing a way for designing will be designing a way of filtering, like the above. In the end, the speaker emphasizes the advantage of conciseness in a language and remarks the clarity of the truth it brings.

#### 1-2 Opinions

I have used programming languages many times, and a variety at that, but never have I thought about designing one. In the flood of diverse programming languages, language was always given for granted. However, struggling through CS320, implementing language comes as one of the amazing fields in computer science. After listening to the speech, I realized that what I was learning in CS320 was one of the fundamental patterns of designing a language, hence, “designing a way of designing”. Lecture about growing a language (4/29) was about finding the limit of a base language and implementing new features on the base language to overcome the limit. In a way, this is growing a language, and it is being done still now by many users out there. The steps I am learning now in CS320 seems to be the most basic steps in designing programming language. In a way I am learning the blueprint of all architecture, cathedral and bazaar included. It is exciting and both frustrating to expect what is to come beyond the basic steps.

### 2. Feature in Real-Language: While statement of Python 3.8

#### 2-1. While statement of Python 3.8

Python was developed in the 1980s and is one of the most popular programming languages currently being used by programmers. It is also a growing language, and the most recent stable version is Python 3.8 (stated from now on as ‘Python’). One of the known features in Python is the ‘While statement’. It can be used for iteration over a block of code as long as the given condition is satisfied, and can be seen as a loop in python. Loops are used to repeat a specific block of code. In Python, there are loops such as the ‘for loop’ and the ‘while loop’. The basic format of While statement in Python looks like this:

```
while <expr>:  
    <statement(s)>
```

<statement(s)> is noted with indentation and represents the block of code to be executed repeatedly-often referred to as the body of the loop- as long as <expr> is evaluated as the Boolean “true”. Initially, <expr> is evaluated as a Boolean context, and if it is true, the <statement(s)> is executed, and <expr> is checked again. If it is true, <statement(s)> is executed once more. This is repeated until <expr> evaluates as false, in which the program execution moves on to the first line of code beyond the body of the while loop. The core role of this feature is to execute the body within the while statement as long as the condition is satisfied. There are additional expressions such as ‘break’ and ‘continue’. However, such expressions are not implemented in the interpreter of this essay,

because it does not play a core role in the feature when using the language.

## 2-2. Code examples of While loop in Python

```
1 >>> n = 5
2 >>> while n > 0:
3 ...     n -= 1
4 ...     print(n)
```

This is an example of the code of the while statement used in Python. Below is the result:

```
4
3
2
1
0
```

n is initially 5, and the condition in the while statement header on line 2 is 'n>0', which is evaluated as Boolean value true, and the body within the while statement (noted with indentation) is executed, which is line 3 and 4, and n becomes 4. When the execution of the loop is finished, program execution returns to the expression 'n>0' in line 2 and it is evaluated again. It is still true, and the body (line 3, line 4) is executed once more. This repeats until n becomes 0, where 'n > 0' on line 2 evaluates as Boolean value 'false'. Since the condition of the while statement is not satisfied, the program execution moves on to the line beyond the while loop. In this case, there is no line beyond the loop and the program terminates.

The condition in the while statement header can differ so long as it can be evaluated as Boolean value. Lists that are empty are evaluated as false, while list with one or more elements are evaluated as true.

## 2-3. Online links about While statement in Python:

In the below links, syntax of While statement and its uses are explained. There are also expressions such as 'continue', 'break'. However, although such expressions help the use of While statement, it is not implemented in the interpreter of this essay. This is because such expressions do not play the core function of the While statement.

[https://docs.python.org/3.8/reference/compound\\_stmts.html#while](https://docs.python.org/3.8/reference/compound_stmts.html#while)

<https://www.programiz.com/python-programming/while-loop>

<https://realpython.com/python-while-loop/>

## 3. Implementing While statement to MFAE

### 3-1. Justification of benefits of While statement – limitations of MFAE

The primary reason that While statement is used in current programming languages- although it differs by language- is to repeat a specific code block until a certain condition is not satisfied anymore. This is advantageous in the situation where you want consecutive actions to be done over and over again. While statement saves a lot of effort for implying the code redundantly. Also, the number of redundancies of the specific code block (referred to as the 'body') is easily controlled because the body is executed only when the condition given in the While statement header is satisfied. This is also helpful when the user do not know when the condition will or will not be met during program execution.

The language 'MFAE with call-by-value' (called MFAE from now on) is a language used in Lecture "Variables". It is BFAE implemented with Variables. It will be the base language for implementing While statement. However, when it is used as a base language, the three features related to 'Boxes'- constructing a box, setting the value inside the box, and getting the value inside the box- will be eliminated because of its lack of use during the explanation. For the same reason, syntax and semantics related to sequence will also be removed. The abstract syntax, operational semantics and interpreter of MFAE used as a base language are as below.

## -abstract syntax

$e ::= n$		
$e + e$	Value $v ::= n$	
$e - e$	$\langle \lambda x. e, \sigma \rangle$	
$x$		
$\lambda x. e$	$n \in \mathbb{Z}$	Environment $\sigma \in Id \hookrightarrow \text{Address}$
$e e$	$x \in \text{Var}$	Store $M \in \text{Address} \hookrightarrow \text{Value}$
$x := e$		

## -operational semantics

$\sigma, M \vdash n \Rightarrow n, M$	
$\frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 + e_2 \Rightarrow n_1 + n_2, M_2}$	$\frac{x \in \text{Domain}(\sigma) \quad \sigma(x) \in \text{Domain}(M)}{\sigma, M \vdash x \Rightarrow M(\sigma(x)), M}$
$\frac{\sigma, M \vdash e_1 \Rightarrow n_1, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\sigma, M \vdash e_1 - e_2 \Rightarrow n_1 - n_2, M_2}$	$\frac{x \in \text{Domain}(\sigma) \quad \sigma, M \vdash e \Rightarrow v, M'}{\sigma, M \vdash x := e \Rightarrow v, M'[\sigma(x) \mapsto v]}$
$\sigma, M \vdash \lambda x. e \Rightarrow \langle \lambda x. e, \sigma \rangle, M$	
$\frac{\sigma, M \vdash e_1 \Rightarrow \langle \lambda x. e, \sigma' \rangle, M_1 \quad \sigma, M_1 \vdash e_2 \Rightarrow v_1, M_2 \quad a \notin \text{Domain}(M_2) \quad \sigma'[x \mapsto a], M_2[a \mapsto v_1] \vdash e \Rightarrow v_2, M_3}{\sigma, M \vdash e_1 e_2 \Rightarrow v_2, M_3}$	

## -interpreter

```

trait Expr
case class Num(num: Int) extends Expr
case class Add(left: Expr, right: Expr) extends Expr
case class Sub(left: Expr, right: Expr) extends Expr
case class Id(name: String) extends Expr
case class Fun(param: String, body: Expr) extends Expr
case class App(fun: Expr, arg: Expr) extends Expr
case class Set(field: String, expr: Expr) extends Expr

// environment
type Addr = Int
type Env = Map[String, Addr]
type Sto = Map[Addr, Value]

// value type
sealed trait Value
case class NumV(n: Int) extends Value
case class CloV(param: String, body: Expr, env: Env) extends Value

def numVop(op: (Int, Int) => Int): (Value, Value) => NumV = (_, _) match {
  case (NumV(x), NumV(y)) => NumV(op(x, y))
  case (x, y) => error(s"not both numbers: $x, $y")
}
val numVAdd = numVop(_ + _)
val numVSub = numVop(_ - _)

def lookup(x: String, env: Env): Addr =
  env.getOrElse(x, throw new Exception)

def storeLookup(a: Addr, sto: Sto): Value =
  sto.getOrElse(a, throw new Exception)

def malloc(sto: Sto): Addr = (-1 :: sto.keys.toList).max + 1

```

```

def interp(e: Expr, env: Env, sto: Sto): (Value, Sto) = e match{
  case Num(n) => (NumV(n), sto)
  case Add(l, r) =>
    val (lv, ls) = interp(l, env, sto)
    val (rv, rs) = interp(r, env, ls)
    (numVAdd(lv, rv), rs)
  case Sub(l, r) =>
    val (lv, ls) = interp(l, env, sto)
    val (rv, rs) = interp(r, env, ls)
    (numVSub(lv, rv), rs)
  case Id(x) => (storeLookup(lookup(x, env), sto), sto)
  case Fun(p, b) =>
    val cloV = CloV(p, b, env)
    (cloV, sto)
  case App(f, a) =>
    val (cloV(x, b, fEnv), ls) = interp(f, env, sto)
    val (v, rs) = interp(a, env, ls)
    val addr = malloc(rs)
    interp(b, fEnv + (x -> addr), rs + (addr -> v))
  case Set(x, e) =>
    val (v, s) = interp(e, env, sto)
    (v, s + (lookup(x, env) -> v))
}

```

In MFAE, let's say we want to add Num(2) to Num(0) until the result of addition is bigger than Num(5). To enable this, we would have to add Num(2) to Num(0) for 3 times. Code following the interpreter would be like this:

```
interp(Add(Num(2), Add(Num(2), Add(Num(2), Num(0))))), Map.empty, Map.empty)
```

The result would be (NumV(6), Map.empty)

```
scala> test(interp(Add(Num(2), Add(Num(2), Add(Num(2), Num(0))))), Map.empty, Map.empty), (NumV(6), Map.empty))
PASS [console>:15]
```

Note that we have to calculate the number of additions before we write the code, which is done by the user, not the interpreter. This type of coding is too much of an effort when the number we want to add gets bigger. For example, if we want to add Num(3) to Num(1) until it is bigger than Num(1000), it is both frustrating to have to calculate the number of additions we have to apply, as well as actually writing the code for addition for 334 times.

Note that the actions for the examples are repetitive. For instance, when we want to add Num(3) to Num(1) until it is bigger than Num(1000), checking whether the number we have to add Num(3) to is bigger than Num(1000), and adding Num(3) to it when it is less than Num(1000) is being repeated 334 times.

Such repetitive actions can be reduced to a simpler code when we implement the While statement. As explained above, the While statement executes a specific block of code (referred to as the “body”) repeatedly when a certain condition is satisfied. In this case, the condition is whether the number we have to add Num(3) to is bigger than Num(1000), and the body is adding Num(3) to the number. The While statement checks the condition after every iteration, and terminates when the condition is no longer satisfied.

### 3-2. Implementing While statement to MFAE

Let us call MFAE with While statement “MFAEW”. But before actually implementing the While statement to MFAE, there are several operations and Value classes added to MFAE to sufficiently use the While statement. As explained above, the model of While statement is as below:

```
while <expr>:
  <statement(s)>
```

In which, <expr> is evaluated as Boolean. However, there are no Boolean values in MFAE. Thus, we need to implement Boolean values. Also, to solve the limitations of MFAE mentioned in 3-1, we imply comparison operators to MFAE. Implementing such operations is not crucial and is not related with the role of While statement, but it helps MFAE to function as the language we wanted by making it easier to use the while statement. Additionally, Null Value is implemented. The reason is explained below.

the semantics of While statement in most of the real languages are as below when expressed in Small-step operational semantics:

$$\langle \text{while } b \text{ do } c, \sigma \rangle \longrightarrow \langle \text{if } b \text{ then } (c; \text{while } b \text{ do } c) \text{ else skip}, \sigma \rangle$$

The  $b$  represents the condition, and  $c$  the body in the above.  $\sigma$  represents the environment While statement is being evaluated upon. Which is to say, ‘while  $b$  do  $c$ ’ is executed on top of the environment that  $c$  is executed in the form a loop when  $b$  is true. When  $b$  is false, the While statement is skipped altogether, and the program execution moves on to the first line of code beyond the While statement. In order to implement ‘skip’ in the interpreter of MFAE, Null Value is implemented. The core of the While statement is repeating a certain block of code as long as certain conditions are satisfied. Thus, While statement is essentially a loop, and in MFAEW, the value of an evaluated While statement is Null. However, the store that maps Address to Value changes depending to the body of the While statement.

The implemented code, namely “MFAEW” will have the following abstract syntax and operational semantics.

-abstract syntax

$e ::= n$ $  e + e$ $  e - e$ $  x$ $  \lambda x. e$ $  e e$ $  x := e$ $  b \quad (\text{boolean})$ $  e = e \quad (\text{equal-to})$ $  e < e \quad (\text{less-than})$ $  \text{while } e \ e \quad (\text{while statement})$ $  \text{Null} \quad (\text{null})$	$\text{Boolean } b ::= \text{true}$ $  \text{false}$	
	$\text{Value } v ::= n$ $  \langle \lambda x. e, \sigma \rangle$ $  \text{Null}$	
	$n \in \mathbb{Z}$ $x \in \text{Var}$	$\text{Environment } \sigma \in \text{Id} \mapsto \text{Address}$ $\text{Store } M \in \text{Address} \mapsto \text{Value}$

-operational semantics – Below is only semantics that was added newly to MFAE. The original semantics of MFAE is the same in MFAEW.

$$\begin{array}{c} \delta \vdash \text{Null} \Rightarrow \text{Null} \\ \delta \vdash b \Rightarrow b \end{array} \quad \frac{\delta, M \vdash e_1 \Rightarrow n_1, M_1 \quad \delta, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\delta, M \vdash e_1 = e_2 \Rightarrow n_1 = n_2, M_2}$$

$$\frac{\delta, M \vdash e_1 \Rightarrow n_1, M_1 \quad \delta, M_1 \vdash e_2 \Rightarrow n_2, M_2}{\delta, M \vdash e_1 < e_2 \Rightarrow n_1 < n_2, M_2}$$

$$\frac{\delta, M \vdash e_1 \Rightarrow \text{true}, M_1 \quad \delta, M_1 \vdash e_2 \Rightarrow v_1, M_2 \quad \delta, M_2 \vdash \text{while } e_1 \ e_2 \Rightarrow v_2, M_3}{\delta, M \vdash \text{while } e_1 \ e_2 \Rightarrow v_2, M_3}$$

$$\frac{\delta, M \vdash e_1 \Rightarrow \text{false}, M_1}{\delta, M \vdash \text{while } e_1 \ e_2 \Rightarrow \text{Null}, M_1}$$

-The interpreter of MFAEW is in Midterm.scala and package.scala.

### 3-3 Overcoming the limitation of MFAE

```
//test case 1
test(interp(App(Fun("x", While(Lt(Id("x"), Num(3)), Set("x", Add(Id("x"), Num(2))))), Num(0)), Map.empty, Map.empty), (NullV, Map(0->NumV(4))))
//test case 2
test(interp(While(Lt(Num(5), Num(4)), Num(3)), Map.empty, Map.empty), (NullV, Map.empty))
```

The above is some of the test cases using the While statement.

Test case 1 is applying a function with parameter “x” and the function body a While statement. The condition of the While statement is ‘x < Num(3)’. The body of the While statement is ‘x:= x + 2’. The argument for parameter “x” is Num(2). Thus, While statement in the function body is iterated twice and the Value that is stored in Sto (type Sto = Map[Address, Value]) mapped with address 0 (address 0 is mapped with “x” in Env (type Env = Map[String, Addr])) becomes Num(4) after While statement is evaluated. In MFAE, in order to achieve the same effect, the user must imply Add(Num(2), Id(“x”)) for two times, and every time after implying one Add(Num(2), Id(“x”)), the user must check whether “x” is mapped to an address that is mapped with a bigger value than Num(3). This is much more complicated compared to using While statement. The value of the expression that is the evaluation of a While statement is NullV, following the operational semantics of MFAEW.

Test case2 is a case when the condition of the While statement is false. The value of the expression itself is NullV, and Sto is the same with the Sto after the condition of While statement was initially evaluated.

### 3-4 Limitations of MFAEW

MFAEW does not implement the complete version of While statement used in real programming languages. For

one, it does not clearly return a value that can be used to check whether it actually changed the environment. Also, value of  $\text{Id}("x")$  in the above test case1 is not checked directly, but through the mapped value of address of  $x$  in 'Sto'. These two limitations were the biggest concern of implementing the While statement in MFAE. However, while implementing the While statement, the primary focus was two things: executing the body only when the condition is true, and the change of environment (in the case of MFAEW, it means  $\sigma$  and  $M$  in abstract syntax) when the body is implemented. In order to satisfy the first point and execute the skipping of the While statement itself when the given condition was false (refer to the operational semantics of While statement in Python), I had to set the value of While statement as 'null' when the condition was false. Also, when the condition is true, While statement would loop repeatedly until the condition becomes false, consequently converging to the case when condition is false. This led to an operational semantics that makes the value of While statement always a 'null'.

However, I decided that the lack of evaluated value of While statement in MFAEW seems a limitation only because MFAEW is too small. In Python, various operations can be used inside the body of While statement, and the two points I mentioned above that I thought as the primary role of While statement would be enough. The limitation that we cannot check the changed environment after While statement is executed in MFAEW will be overcome if other operations that enable it is added. However, I added this part in the essay because although this was the best I could do during limited amount of time, it would have been better to write a more clarified operation that would let MFAEW easier to use.