COMP 302 Study guide

Francis Piche

March 15, 2018

Contents

Part I
Scala Basics

1 The Scala REPL

The Scala REPL (Read-eval-print-loop)is opened by typing ”scala” into your command line (given Scala is installed)

It’s basically a place where you can get instant feedback on pieces of Scala code.

Play around and get a feel for it! When something works, you can copy it over to a text file. (or even the other way around!)

2 Basic Expressions, Types

Typing lines of code and pressing enter gives a result. This result is given a variable name which we can use later.

Example:

```
scala> 1 + 1 //your input, followed by enter.
res0: Int = 2 //the output. The variable res0 now contains the
   integer 2.

scala> "Cool guide bro " + res0
res1: String = Cool guide bro 2 //demonstrating using stored
   variables.
```

We can also use our own variable names with the `val` command. This creates a *constant* that can be used instead of the default one. As a constant, it can’t be changed later.

```
scala> val foo 1+1
foo: Int = 2 //instead of res0 we see "foo"
```

In contrast we have the `var` command, which creates a changeable variable.

```
scala>var foo 1 + 1
foo: Int = 2
scala>foo = 2
```

To declare types, which is not always necessary, the syntax is:

```

scala> val test: String = "bar" //type comes after variable name

scala> val test0, test1,test2 = 50 //you can also declare several
   at once
```

The common types are:

- Byte
- Char
- Short
- Int
- Long
- Float
- Double
- String

In Scala, there are no primitive types. All types come from classes.

Note that the String class uses the same library as Java, but with some extras such as the StringOps class.

There is no casting between numeric types. Instead use methods such as `toInt` or `toChar`.

Also note that the operators * / + - = etc are actually methods themselves.

You could write `5.+(2)` instead of 5+2 if you wanted to. In general, methods are used like this: `arg0 method arg1` or `arg0.method(arg1)`. Note that if a method has no parameters, you don't need to include the '()'. You can simply write "arg.method".

THERE ARE NO INCREMENT/DECREMENT OPERATORS. (++, –) Use +=, -= instead.

Whenever using a class you haven't before such as BigInt or the util package, be sure to read up on the documentation.

3 Control Structures

3.1 The If Statement

The `if` statement in Scala is a combination of the `if` statement and ternary operator from Java/C. It evaluates to a value and type, but can also include multiple statements.

For example:

```scala
val x = if(5>2) 1 else -1
```

Here the type Int was inferred, but if you have:

```scala
val x = if(5>2) "potato" else -1
```

No type can be inferred and so the Any type is used.

If you omit the else, something like:

```scala
val x = if(5>2) 1
```

And the condition is not met, then there is no value, but it must return a value so it returns Unit (). Which is basically a "nothing" value.

3.2 Blocks

If you have a block of code (code in curly brackets) then the last line is what the expression evaluates to. (Since everything evaluates to a value).

Note that value assignment has Unit() value.

3.3 Loops

`do` and `while` loops are the same as in Java/C

`for` loops are quite different.

Scala tries to minimize the updating of variables, and prefers to stick to values. So for that reason, the concept of ”incrementing” is not involved. Instead, you pick a local variable, and use it to iterate over a fixed collection.

The `for` loop:

```
for(i <- expression)
```

where the expression can be something like a string, or the integers 1 to 10.

You can actually have multiple expressions in the loop header, separated by a semicolon ;.

You can also put `if` statements in your loop header to have it only iterate one thing if a certain condition is met or not met.

If the body starts with `yield` then the loop will return a collection of your yield statement(s).

4 Functions

A function is not a method. A function doesn't operate on an object.

Here's a basic method:

```
def foo(x: Double) = {
```

```
//body
//the last statement is the return value
//return type is implied (unless function is recursive)
}
```

The `def` keyword marks it as a function (short for define), followed by the name of the function, and in parenthesis the variable input, and it's type. Here "function" is like saying $f(x) = stuff$ in a math-context.

If the function is recursive you should specify the return type like:

```
def foo(x: Double): Int = {

   s = s + foo(x/2) //random recursive call
}
```

4.1 Variable arguments

If you don't know how many arguments your function will take, you can use:

```
def foo(args: Int*) = {

   //args is now a collection of the arguments you gave
}
```

Note that if you pass a collection as an argument, you must use: _* to denote it as an argument sequence.

```
val x = foo(1 to 10: _*)
```

4.2 Procedures

A procedure is a function that returns type Unit (). Recall Unit() is a value meaning "no value". These are functions that do things like printing. You can either specify the return type Unit in the header or just remove the "=" from the header.

5 Arrays

5.1 Fixed-Length Arrays

Initialize an array like this:

```
val x = new Array[Int](10)
//array of type Int of length 10
y = x(0) // this is how you access indices
val z = new Array("One","Two","Three")
//creates an array of length three, and type String is inferred.
```

Note that while the Array is a val, and cannot be changed, the elements are mutable.

5.2 Array Buffers

These are basically ArrayLists from Java.

```
import scala.collection.mutable.ArrayBuffer
val x = ArrayBuffer[Int]()
x+= 1 //adds element 1 at end
x += (1,2,3,4,5) //adds all these at end
x.insert(2,6) // insert number 6 before index 2

x.toArray //converts to array
x.toBuffer // converts to buffer
```

5.3 Traversing Arrays

You can traverse arrays using a for-loop since they are a type of collection.

```
for(i <-until x.length) //to traverse over the array (avoiding
   index out of bounds errors)
for(i <- 2 until x.length by 2) //starting at index 2 and going by
   2's
for(i <- 2 until x.length by -1) //starting from end, going
   backwards to 2.
```

6 The Match Statement

In Scala, we have something similar to the `case` statement in other languages. But this one doesn't have any "fall-through", so break statements are unnecessary.

```
x match {
   case 0 => do something
   case 1 => do something else
   case _ => default case
}
```

Here, x is the parameter. And different things happen for values 0, 1 and anything else.

You can also include conditionals in the case, or even check for types!

```
x match {
   case s: String => "It's a string!"
   case i: Int => "It's an Int!"
   case (t1, t2): "It's a tuple!"
   case _ => "Not sure what it is..."
```

You can also match collections!

```
   a match {
      case Array(0) => "0" //matches the array containing 0
      case Array(x, y) => do something//matches the array with two
          elements
      case _ => "any array will do"
```

7 Collections

7.1 Tuples, Lists

A tuple is a pair, triple, quadruple, 5-tuple and onward. (Basically an ordered list of things)

```
val t = ("balloon",5,2.0) //creates a 3-tuple
val t._1 //gives 1st element of the tuple. ("balloon")
```

Tuples have a size limit (quite small).

We can take them apart like so:

```
val (s,i,d)= t //assigns s = balloon, i = 5, d =2.0
```

Lists are nicely made linked-lists. They end in a "Nil" List object which indicates the end.

A List is created like so:

```
val grades = List("A","A-","B+","B-","C+","C","C-","D","F")
\\ or alternatively

val grades = "A"::"A-"::"B+"::"B-"::"C+"::"C"::"C-":::"D"::"F"::Nil

grades.head //returns first element
grades.tail //gives last element
grades.tail.head // gives second element, as follows by the
   recursive definition
```

Note that a List can be recursively defined by:

$$Head|List$$

Meaning that the List is a head node, followed by a sub-List.

7.2 Maps

A map associates a key to a vale. For example:

```
val phonebook = Map("Henry" -> 1234567, "Bob" -> 9119999)
phonebook("Henry")//returns 1234567
```

We update this immutable Map by creating a new Map with the desired changes.

7.3 Iterating over Maps

You can use for-yield to perform an operation on a map

```
def timesTwo(grades: Map[String, Double]): Map[String, Double] = {
   for((name,mark) <- grades) yield (name,mark*2)
}
```

This multiplies all marks in the Map by two.

8 The map operator

Can use this on any collection in Scala. It processes each element of the collection. (Applies a function on each element)

For example:

```
val s = 1::2::3::4::Nil //creating a list
s.map((x) => x+1) //increments each thing in the list by 1
```

Note that map expects a single argument function, so say you had a Map and wanted to do things to the individual elements, you'd have to do something like this:

```
val m = Map("key" -> 1, "key2" -> 2)
m.map((x) => ((x._2, x._1))) //x is a single argument, but is in
    fact a tuple.
```

The above function swaps the keys for the values in Map m.

9 Reducing and Folding

The `reduceLeft` function applies an operation in a cascading way. For example:

```
(1 to 5).reduceLeft(_+_)
```

Applies the function _+_ to the list, basically taking each element,and adding it to the next, *accumulating* along the way. So first 1 + 2 = 3. Then 3 + 3 = 6, then 6 + 4= 10, then 10 + 5 = 15. So this statement would return 15.

The mkString function is actually a type of reduceLeft, it works like this:

```
(1 to 5).reduceLeft((s1,s2) => {s1 + "," + s2})
```

However the above gives an error. This is because at first, an Int is concatenated with an Int to give a String. Then an Int is concatenated with String, and so on. So at the end, the result is a String, but the input was a list of Ints. So Scala gets mad and throws an error.

To fix this, there's a few things we could do.

```
(1 to 5). reduceLeft((s1: Any,s2) => {s1 + "," + s2}) //now types
    work but returns type Any

(1 to 5).map(_.toString).reduceLeft((s1,s2) => {s1 + "." + s2})
    //uses the map function to turn each Int into a string first
```

Note that reduceRight also exists, which just starts from the right of the list instead of the left.

The `foldLeft` function allows you to specify a starting value, like this:

```
(1 to 5).map(_.toString).foldLeft("!")((s1,s2) => {s1 + "," + s2})
//notice the brackets: foldLeft()()
//This is because foldLeft takes 1 argument then returns a
    function which takes another argument.
```

In general, folding can be used to iterate through a List, applying a function and accumulating as we go along.

```
(Map[Char, Int]().foldLeft("Mississippi")((m,c) => {m + (c ->
    (m.getOrElse(c,0)+1))})
```

This takes an empty map, takes the String "Mississipi" as a starting value. (m,c) are the arguments for the function we're going to apply to the Map. m is the map itself, and c is the current character. Then, we add to m, the entry (c -¿ (m.getOrElse(c,0)+1)), where c is the character key, and (m.getOrElse(c,0)+1 gets the value for the key c from the Map m, if it exists. If it doesn't exist, it gives 0. We add one to this value to increment.

Wow that was some heavy syntax.

Overall, what this does is create a Map that keeps track of the occurrences of each character in the word "Mississippi".

```
val s = 1::2::3::4::5::6::Nil
(s.foldLeft(false)((b,i) => {if (!b && i==3) true else b})
```

This looks for the number 3 inside the list s by folding. It takes false as a starting value. b is a boolean that keeps track of whether we've seen a 3. i keeps track of which number we're at. If we find 3 and we've not seen it before, then we'll output true. If not, then we'll output what we went in with, b. (If we've seen it, b==true).

10 Miscellaneous Useful Things

10.1 File IO, String functions and other functions

```
import scala.io.Source
val file = Source.fromFile("filenamehere.format", "UTF-8")
//creates file object you can iterate over

val text = file.mkString //makes a string of the entire file

val words = text.split("\\s+") //regex for whitespace
//makes an array of every word separated by whitespace.

val lowerCase = for ( w <- words if (w(0).isLower)) yield w
//takes every word in words that starts with lower case letter

val caseless = for (w <- words) yield w.toLowerCase
//converts all words to lower case

val c2 = caseless.clone //makes clone of caseless
```

```
scala.util.Sorting.quickSort(c2)
//sorts c2 (not functional)
//changes actual Array elements of c2

val sorted = caseless.sorted
//sorts caseless without changing original (functional)
```

Part II
Functional Programming

In functional programming, we don't update data. Only create new versions.

Iterating is done by recursion.

11 Recursion

11.1 Head vs. Tail Recursion

In **head recursion** is when you recurse all the way down to the base case before returning up to the top. Computing happens "on the way up". For example, consider the factorial function:

```
def fac(n: Int):Int = {
  if(n=1) 1 else fac(n-1)*n
}
```

This will call fac(n-1) until the base case is reached, **then** start multiplying. Whereas with **tail recursion** we *accumulate* the result along the way.

For example:

```
def tailfac(n:Int, m:Int): Int m = {
  if(n==0) m else tailfac(n-1, n*m)
}
```

This function computes **before** the recursive call.

This idea was used in assignment 1 for the Fibonnaci sequence. (So tail recursion is sometimes more efficient!)

Note that we can avoid the two parameters by nesting tailfac(n,m) inside another function that takes only one parameter, (n) like so:

```
def tailfac(n: Int): Int m= {
   def tailHelper(n:Int, m:Int): Int m = {
      if(n==0) m else tailHelper(n-1,n*m)
   }
   tailHelper(n,1)
}
```

12 Higher-Order Functions

12.1 Functions are Values Too!

In functional programming, functions are passed around in a similar way to how other data types are.

You can store a function in a variable like so:

```
val ayy = floor_
//ayy contains the floor function.
//the _ specifies you meant a function, and didn't make an error
   in calling it.
```

Now you can call the function using the value name, and can also pass it into other functions like so:

```
ayy(2.5) //outputs 2

Array(1.5,2,3,1.8,1.9).map(ayy) //outputs Array(1,2,3,1,1)
```

Here you can see that the map function takes a function as a parameter and applies it to everything in the collection.

So as you can see,functions can be returned from functions, passed into functions, and be generally treated like a variable.

12.2 Anonymous Functions

An anonymous function is a function without a name.

Example:

```
(x: Double) => 3*x
//alternatively:
def multby3(x: Double) = 3 *x
//does the same thing.
```

You can pass this into other functions like we did in the last subsection.

```
Array(1.5,2,3,1.8,1.9).map(x: Double) => 3*x)
//multiplies everything in the array by 3.
```

12.3 Functions as Parameters and Returning Functions

All we need to do is specify the parameters that our parameter function will take, and it's return type. This allows us to leave the implementation up for grabs.

Example:

```
def function(f: (Double) => Double) = f(0.25)
```

This defines a function called function that takes as input a function, f, which takes a Double as input and returns a double.

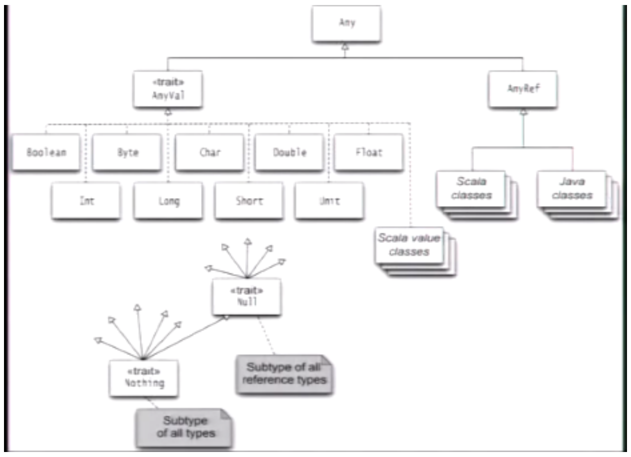
You can also *return* functions from functions like this:

```
def function(f: Int): (String) => String = {
   stuff
}
```

This function returns a function that takes a String and returns a String.

13 Types

In Scala, the type hierarchy is like this:



Notice that Any is a super type of everything. Any is split into AnyVal (kind of like primitive types), and AnyRef (other objects).

Null is a sub-type of all references types. (Contains null value)

Nothing is the sub-type of everything. (Even values!)

Notice that even though Int fits inside Long, it is actually not related at all, although conversions exist.

Unit is a value for void.

13.1 Type Inferencing

A good example of how types get inferenced in different scenarios is with the conditional.

```
val t = if(3>4) 5 else 8 //both 5 and 8 are Int so returns Int
val t:Int = if(3>4) 5 else 8 //specifying exactly what we want
val t = if(3>4) 5 else true //unable to inference, goes to
    super-type of Int and Boolean (AnyVal)
val t = if(3>4) 5 else "Hello" //goes to super-type of Int and
    String (Any)
```

The type Nothing indicates errors/failures. For example, exceptions have type Nothing. Allows a function like this to still return Int:

```
def divide(x:Int, y:Int): Int = {
   if(y!=0) x/y
   else throw new Exception....
}
```

14 Scope

The scope is the area in which a set of bindings is active.

14.1 Local Variables

Classically (mathematically) we'd define a "variable" as:
Let $x = \_\_ \in V$ where V is some space.

Consider:

$$f(x,y) = x(1+xy)^2 + y(1-y) + (1+xy)(1-y)$$

Notice there's some redundancy here. We $(1+xy)$ and $(1-y)$ are repeated. So we could either declare local variables, or we could do this:

```
def f (x:Double, y:Double) = {
   def fHelper(x:Double, y:Double, a:Double, b:Double) = {
      x*a *a + y*b + a*b
   }
   fHelper(x,y,1+x*y,1-y)
}
```

This does exactly the same as our math expression above. It eliminates the need for local variables (and eliminates the redundancy) by passing what would have been local variables as the arguments to a function.

This is called ”Let-Abstraction” instead of creating local variables, we define a nested-function which takes a new parameter, which will be used by the outer function to simulate a local variable.

Just goes to show, all we ever need is functions.

14.2 Lifetimes

A variable is defined until the end of the function in which they were defined, or if they are global, until the end of the program.

So, if you define a function using val = syntax, it exists in this way.

When you define a function using `def` syntax, it gets a different lifetime, so that you're able to do recursion.

Values also have lifetimes. (Memory is needed for values)They exist until we free that memory,either manually or by garbage collection. (Depending on the language)

14.3 Bindings

We associate names with values in scopes by binding the value to the name. In functional programming, we try to avoid re-binding. A set of bindings defines the program state.

The bindings we can access are the ones within the scope.

Explicit bindings are when we specifically declare a variable. `val x = 1`.

Implicit/Indirect bindings are when we define something within a function definition. `def foo(a:Int) = ...` We associate parameters with arguments.

Sets of bindings form an environment.

Whenever we mention x, we have to go into the environment and figure out what x is bound to. For example when we do x=z we have to evaluate z, (look up the right-hand side in the environment). We then look up the binding of the left-hand side, and rebind the value of z to x.

14.4 Trade-Offs and Choices

When designing a language, we have to decide "what makes sense". For example:

```
val y = 2
def foo(x:Int) = {
  x+y
}
//sequential adding and manipulating the environment (single pass)
foo(3)
val y =2
def foo(x:Int)= {
  x +y
}
//non sequential (double pass)
```

Both of these "make sense" in most languages.

```
val y = z
val z = 8
```

Makes sense, but depending on the language might decide to make it not work.

14.5 Static Scope

We don't need to know the run-time control flow, we can simply look at the code and know which variables can be accessed where. (no dynamic runtime info needed).

The simplest model of this is having **one scope**. No procedures/functions. Everything is global. So the environment contains every single variable and value in the entire program.

This is quite prone to errors in larger projects

Local + Global Scope: the scoping we're all familiar with.

We can have sub-scopes within our local scopes (nesting). If we declare a variable inside a sub-scope, we lose the access to the outer scope variable.

Shadowing is when this inner-scope re-naming occurs. Some languages do not allow this (not common).

14.5.1 How Does It All Work?

It will work by nesting (or chaining) of environments. Where one environment points to it's "parent" environment. If the variable we're looking for isn't in the current environment look "up" to the parent.

Suppose we have:

```
var x = 10
{
  var x =1
    x = x+x
}
x=x+x
x
```

What happens? Well first our current environment is set to contain the binding `x:10`, next, we create a new environment (since we're moving inside a new scope defined by curly brackets) we set the new environments parent to be the current, and the new one is now the current environment.

Now, the binding `x:1` is created inside the newest environment. Then, x is updated, so the binding is changed to `x:2`.

Now we leave that environment and return to the parent. Here, x is 10. So `x=x+x` gives the binding `x:20`. Finally, we return 20.

14.5.2 Declaration Order

14.5.3 Modula-3 Style

Declaration order doesn't matter. (This is how Scala works).

However, notice this strange behaviour in Scala:

```
val y = 8
def foo() = {
   val x =y
   val y=2
}
```

This gives an error. What's happening is that at the time that foo is defined, we have two environments. One containing `y:` and another containing `x:`, `y:`. While the declarations are unordered, the *assignments* are. So it will try to assign y to x by looking for y in the current scope. It exists, but it currently has no value! This is what results in the error.

Why unordered?

Mutually recursive functions! If we needed to define a function before it is invoked, there would be no way to do this.

Would it ever make sense to have something like:

```
val x = y
val y = x
```

Not generally, but if we think of sets, we might want to do something like $X = Y \cup a$ and $Y = X \cup b$. In that sense, it does "make sense"

Suppose we have:

```
var n =0
def X() = {
   n=1
}
def Y() = {
   var n =2
   X()
}
X()
```

Whats happening? We have a global environment containing `n:, X:<func>, Y:<func>` then, when we call X, we create a new environment, and set it's parent to the environment in which it was defined (the global one in this case). So inside X, we have access to all of X, and the global environment (so we can update n).

Suppose now:

```
var n =0
def X() = {
   n=1
}
def Y() = {
   var n =2
   X()
}
Y() //new!
```

We start off the same way, with the same global environment. But when we call Y(), we create a new environment for Y, in this new environment we have `n:2`. Then we call X(), which was defined in the global scope, so we create a new environment with it's parent to be the global scope. We set n to 1 as in the first example, and then throw away the X() environment (we're done with it now). We throw away Y() since we're done with it, and the resulting change is that the global n is set to 1.

The important thing to notice is that we didn't have a straight line of references. X() did not have Y() as it's parent environment, even though it was called from Y().

14.5.4 The Essence of Static Scoping

The parent environment is the scope in which the current was **defined**.

The current environment is part of a stack that follows the function calls.

To look up a symbol, we start at the current environment, and go up the parents to find it.

14.6 Dynamic Scoping

Here, the parent pointer is based on the **current** environment (where the func is called) (Not where the function was defined).

```
var n =0
def X() = {
   n=1
}
def Y() = {
   var n =2
   X()
}
Y()
```

Here, when we call X() from within Y(), X's parent is now Y instead of the global environment. So when we lookup n, we look to Y(), not global. So when we say n=1 within X, it's changing the n inside Y, not the global.

Dynamic scoping seems like it adds lots of flexibility, but prone to errors.

Example:

```
def foo() = {x}

def bar() = {
   foo()
}
```

If foo is called from the global scope, it should produce an error, if it was called from bar(), it would work. So it becomes very messy to figure out what's actually happening in the code.

14.7 Parameters

Parameters live inside the function, so they're kind of like local variables.

```
var n =0
def X(a:Int) = {
   n=a
}
def Y(a:Int) = {
   var n =a+1
   X(n+1)
}
n=3
Y(n+8)
```

Here, when Y is called, a gets $n + 8$ from the global scope, so $a = 11$. Now within Y, we define a new n, so within Y, $n = 12$. Next, when we call X(n+1), we bind a in X to 13. Within X, we set the global n to 13.

When we evaluate arguments, we evaluate them in the current environment, then pass them into the new environment.

14.8 Nested Functions

If we have nested function, it still works in our static scoping paradigm. The parent of the nested function is just the function in which it was defined.

14.9 Recursion

```
def fac(n: Int) : Int = {
   if(n==0) 1
   else fac(n-1)*n
}
fac(2)
```

Fac is defined in the global environment. When fac(2) is called, we create an environment for it. It's parent is the global. When fac(1) is now called from within fac(), we create another environment with parent pointing to global. So we end up with a bunch of environments all pointing to the global environment.

14.10 Executing Code

Executing code requires the code itself and the starting environment.

We can then treat executing as a "thing" or an "object".

We've been doing this with functions. Then, we needed the parameters, the body, and the environment. We treated it as a thing. This is officially called a **closure**. (Function name, environment, parameters and code).

A **continuation** is when you package up everything that happens after the function!

This allows us to do things like this:

```
def addMaker(x:Int): (Int) => Int = {
   (y: Int) => {x + y}
}
val add11 = addMaker(11)
add11(3) //returns 14
```

To know everything about addMaker, we need it's content, and the environment it was defined. The environment it was defined was global. Now, when we call addMaker(), we create a new environment, and we know exactly what the parent is, since it's contained in the information for addMaker. The new environment contains the information for the nested anonymous function. add11 is then bound to this new function.

But what about with mutable data?

```
var x = 8
```

```
def incMaker() = {
   () => {x = x+1}
}
val s = incMaker()
x
```

This works exactly as we'd expect. s is a function that does nothing, but points to a function that does x = x+1, who's parent is the global environment. So when we call s(), we increment the global x.

We can use the encapsulation of local data inherent to closure to build objects.

```
def makeObj() = {
 var name =""
 ( () => {name}, (n:Strin) => {name =n})
}
```

This makes an object which is a pair of things, a function that returns the name, and a function that sets the name.

The reason all this is working is because of how the closures work.

If we had something like:

```
def hMaker(){
   var words = Array ("hello", "hi", "yo")
   var helloF = new Array[()=>String](3) //array of functions of
       size 3
   var i =0
   while(i<3) {
      helloF(i) = () => (words(i)) //helloF will return the word at
          words(i)
      i=i+1
   }
   helloF //return the array of functions
}
```

This fails when we call hMaker() for a non-intuitive reason. When we invoke it, we create a new environment, it has local variables words, helloF, and i. When we start iterating, we create new functions. Each get their own environment, who's parent environment will be the hMaker environment. In the last line of the loop, i is incremented to 3. Now, when we try to access helloF(i), it doesn't exist in the environment of helloF(i), so it goes to hMaker, but then i is 3, which is now ArrayIndexOutOfBounds.

We could avoid this by each function in helloF to have their own i.

```
def hMaker2() = {
   val words = Array("hello","hi","yo")
   (for(i<-0 until 3) yield { () => (words(i))}).toArray
}
```

This works because the for-loop gives us a unique environment for each iteration, this new one contains it's own version of i.

Part III
Language Construction

Need to be formal. Need to specify every detail, that leads to implementation. We need to specify the syntax (the languages representation), and the semantics(what it means to execute it).

15 Syntax

Programs are just a mass of symbols, so we have a stream of characters. We want to push this through a compiler or executer to get something that works.

Syntax will be broken down into a "pipeline" of steps. First, we take our stream of chars, and put it into a scanner. From the scanner we'll get a stream of tokens. Next, we'll pass these tokens into a parser. We can either execute this output right away, or compile it to get an executable.

15.1 The Scanner

Chars come in, and we want to generate tokens. These tokens are just things that "mean" something. Keywords, operators, symbols , identifiers, numbers, floating point vs. integers, comments, whitespace, etc. are examples of tokens. Tokens are the meaningful pieces of our language.

The set of tokens is specific to the language.

It's worth noting that there can be overlap in our tokens. For example, `val` is a keyword, but `value` could be used as an identifier. So we can't simply just stop reading once we see `val` and assume we have a token.

We need to then peek ahead to see that the next character is a white space.

15.2 Regular Expressions

Regular expressions are used for matching/finding tokens. So our scanner is really just running multiple regular expressions looking to match our tokens.

Reg Ex. is a language in itself for recognizing patters. It has limited expressiveness.

15.2.1 Definition

A regular expression, (RE) is:

- Empty string, ϵ.
- Single character.
- If R_1 and R_2 are RE's, then R_1R_2 is a RE.
- If R_1 and R_2 are RE's, then $R_1 \mid R_2$ is a RE. (The $\mid$ means OR)
- If R_1 is a RE. Then so is R_1*. (The $*$ is 0 or more repetitions of R_1.

15.2.2 Examples

```
(a | b) //matches "a" and matches "b", doesn't match "c"
```

```
(abc) | (def) //matches "abc", and "def", doesn't match "abcdef"
```

```
(abc)* //matches "", "abc", "abcabc"... forever.
```

```
(abc)(abc)* //matches one or more repetitions of "abc"
```

15.2.3 Extra Syntax

- + means one or more instances.
- ? means 0 or 1 instances.
- [a-z] means all alphabetic lower-case letters.
- [a-z A-Z] means all alphabetic letters.
- [a q w f 3 !] means OR any of the things in the brackets.
- [ˆa-z] any character except a-z
- [\n \t \r] is the same as \s and will take out any whitespace
- {} are for a specific number of repetitions
- be(?!e) finds the instances of be, as long as its not followed by another e.

- See java.util.regex.Pattern for the full guide.

15.2.4 Anchors

The ˆmeans ”match start of string” when used outside of a character class (square brackets).

The $means ”match end of string”

15.2.5 Reg Ex and Scala

To turn a string into a regular expression, we use the `.r` operator.

Example:

```
val numberPattern = "[0-9]".r //numberPattern is now a RE for
   numbers.

numberPattern.findAllIn("abc 123 8 ppp 8") //finds all matches in
   this string. returns an iterator that we can turn into an array
   to see all the matches

numberPattern.findFirstIn("abc 123 8 ppp 8") //finds first match,
   returns as Option of type Some(x) If no match is found, returns
   an Option of type None.
```

Notice findFirstIn gives you the Some object or None object, rather than the string itself.

```
val id = "^[_a-zA-Z][_a-zA-Z0-9]*".r //regex for an identifier
val key_val = "^val\b".r //matches starting with val then ends the
   word. (\b means end of a word)
```

```
val bees = "b+".r
bees.findAllIn("bbbbbbbbb").toArray
```

The above finds only one match, the longest possible. So we wont get a bunch of ”b”, ”bb”, etc.

15.3 Examples of Scanners

```
val re_id = "^[a-zA-Z][_a-zA-Z0-9]*".r //starts with alphabetic
    chars, then any length of alpha-numeric chars.

val re_ws = "^\s+".r //whitespace

def scan(input: String, tokens: List[String]): List[String] ={
   if(input =="") tokens
   else{
      re_ws.findFirstIn(input) match {
         case Some(x) => return
            scan(input.substring(x.length),"WS"::tokens)
         case None =>
      }
      re_id.findFirstIn(input) match {
         case Some(x) => return
            scan(input.substring(x.length),"ID("+ x + ")"::tokens)
         case None =>
      }
   }
}
```

The above code defines two tokens as regular expressions. It then defines a recursive function that uses findFirstIn to match the input string. Remember that findFirstIn returns type Some if it finds something, and None if it doesn't. If it finds something, we append it to the list, and make a recursive call.

It's worth noting that the final returned list will be in the opposite order of the input string, but that's okay since its easy to reverse a list.

15.4 Parsing

A program is just a series of declarations, variable declarations, class declarations, function declarations, etc.

Each declaration is just a series of tokens. For example, declaring a variable has `var ID = VALUE`, functions have `def ID "(" parameter list ")"` `type declaration = expression`.

15.5 Backus-Naur Form (BNF)

This the the grammar system we'll use to build our language.

BNS grammar has a list of rules. Some of these rules might be recursive, or refer to other rules. The rules are composed of symbols. The symbols are broken into two categories: **terminals** (our tokens) and **non-terminals** (our rules).

The structure of a rule is like this:

```
LHS ::= list of terminals + non-terminals
```

Note that every non-terminal appears on the LHS of at least one rule.

15.5.1 Examples of Rules

```
forStmt ::= FOR "(" ID "<-" EXPR ")" YIELD EXPR
EXPR ::= binEXPR | unEXPR | fncall | ifEXPR | ...
or
EXPR ::= binEXPR
EXPR ::= unEXPR
EXPR ::= fncall
...

blockEXPR ::= "{" EXPRlist "}"
exprList ::= EXPR EXPRlist
```

Note that the exprList ends with nothing, denoted ϵ.

So basic BNF gives us non-terminals and terminals, the "|" (or) operator and recursion as our tools. But what if we want to have repetitions, and stuff like regex?

15.5.2 Extended BNF

We now have *(0 or more repetitions), (), ? (optional type) For example:
`argList ::= arg(","arg)*`
`fundecl ::= .....typedef?`

To start off the whole program, we might start with the initial rule of: `program ::= decls*`

15.5.3 Parser Example

Say we want to parse `a = b + c`

```
assignStmt ::= ID "=" expr
expr ::= ID | expr binOP expr
binOP ::= "+" | "-" | "*" | "/"
```

15.5.4 Parsing with Scala

```
import scala.util.parsing.combinator._
class ExprParser extends RegexParsers {
   val Number = "[0-9]+".r
   val Plus = "+" //can mix strings and regex in scala parser
   val Minus = "-"
   val Times = "*"

   def expr : Parser[Any] = term ~ opt((Plus | Minus) ~ expr)
   def term : Parser[Any] = factor ~ rep(Times ~ factor)
   def factor: Parser[Any] = Number | "(" ~ expr ~ ")"
}

val p = new Expr Parser()
val ast = p.parseAll(p.expr, "3-4*5")// returns:
    ((3~List())~Some((-~((4~List((*~5)))~None))))
```

This big string that gets returned is a term (3 List()) with an optional part. Optional things return Some and None. The empty list inside the term is because a term is made up of a factor (the 3) and a repetition of expr. There was no repetition so the List is empty.

Note that whitespace is discarded by default.

The double caret operator at the end of a rule, lets us actually evaluate the AST when it is called.

```
class ExprParser extends RegexParsers {
  val number = "[0-9]+".r
  val Plus = "+"
  val Minus = "-"
  val Times = "*"

  def expr: Parser[Int] = term ~ opt((Plus | Minus)~expr)^^{
    case t ~ None => t
    case t ~ Some(Plus ~ e)=> t+e
    case t ~ Some(Minus ~ e)=> t-e
    case t ~ Some(_) => t //error?
  }
  def term: Parser[Int] = factor ~ rep(Times ~ factor) ^^ {
    case f ~ list => f*list.map((x)=>x._2).product
  }
  def factor: Parser[Int] = number ^^ {_.toInt} | "(" ~ expr ~
     ")"^^{
    case _ ~ e ~ _ => e
  }
}
```

The "factor" rule converts the input String numbers into Ints, or, if it's an expression surrounded in brackets, output just the expression.

The expr rule evaluates the expressions as Ints. If it's just one thing, then it's just a term. Else, evaluate the expression.

The term rule takes a list of factors and applies the map to multiply them all together. We don't include the Times symbol because there's really no reason to keep them. The .product method is just a method that does folding for you.

To now use this parser, we would do:

```
p.parseAll(p.expr, "3*1*4*5*2+1").get
```

Here, we specify the starting rule (expr), the string to be parsed, and we call .get to evaluate it.

15.6 Class Heirarchy

A class heirarchy for our expression parser would look like:

```
abstract class ASTNode
class ASTNumber(val value: Int ) extends ASTNode
class ASTBinOp(val op:String, val left: ASTNode, right: ASTNode)
   extends ASTNode

def expr: Parser[ASTNode] = term ~ opt((Plus | Minus)~expr)^^{
      case t ~ None => t
      case t ~ Some(Plus ~ e)=> ASTBinOp(Plus,t,e)
      case t ~ Some(Minus ~ e)=> ASTBinOp(Minus,t,e)
      case t ~ Some(_) => t //error?
}
def term: Parser[ASTNode] = factor ~ rep(Times ~ factor) ^^ {
      case f ~ Nil => f
      case f ~ list => {
         val values = f::(list.map((x)=> x._2))
         values.reduceLeft((x,y)=> ASTBinOp(Times,x,y))
      }
}
def factor: Parser[ASTNode] = number ^^ {(n=> ASTNumber(n.toInt)}
   | "(" ~ expr ~")" ^^{case _ ~ e ~ _ => e}
```

This does the same as before, but gives us a nice AST representing the expression. It doesn't actually evaluate though.

16 WML

This is the language we'll be building. Inspired by wiki pages, which is a nicer way to represent HTML, in wikiText.

The part of wikiText we're interested in for this class is the template language. The template language sets the framework so that we can create templates that can later be used on other wiki pages.

For example, if we have {{`foo`}}, this means, go find the page called foo, and copy the contents here.

These templates can take parameters. We do this using the pipe operator `|`. For example {{`foo | bar ta ta | fou | `}} where the call to the other template is "foo", and the parameters are "barr ta ta", "fo" and the empty string "".

Within the page where we define the template "foo", we can call on these parameters with triple brackets and indices. {{{`1`}}} would refer to "bar ta ta" and {{{`2`}}} to "fou".

We can also name our variables like so: {{`foo | first = bar ta ta |second = fou | `}}

So now when we want to call them in the template: {{{`first`}}}, {{{`second`}}}

Note that it's not fully functional because does not allow for recursive transclusions. Obviously, we can't copy "foo" from "foo", or else we'll end up in an infinite loop.

There also isn't a real scoping model, since we can access any webpage from anywhere. Theres no way to create or pass functions anonymously, or treat them like variables.

In this class, we'll build a real functional language from this.

16.1 What We'll Build

We wont follow the name assignment syntax of the wiki as mentioned above. Ours will have something like:

```
{{foo | bar | bla}} //calling the template

foo : abc, def //defining foo, takes two parameters, abc, def
{{{abc}}} //calling parameter 1 (bar)
```

We wont have "pages", so we need a special syntax for defining templates. They will all live in the same file.

We'll need a grammar for this. This is what we do in the assignments.

16.2 Building it up from scratch

In this section we'll build up the language from no features to all the features.

First is a language with nothing, just plain text. This will just be ¡program¿ ::= OUTERTEXT

Now what if we allow invocations?

Recall invocations look like: $\{\{foo|arg1|arg2\}\}$. So we would need OUTERTEXT to allow ¡invoke¿.

So now our language looks like:

```
<program> ::= (OUTERTEXT |<invoke>)*
OUTERTEXT = anything, except for TSTART
<invoke> ::= (TSTART <itext> <targs> TEND) //TSTART/END are {{}}
   and itext is the name of the template
<targs> ::= (PIPE <itext>?)* //this creates a list of arguments
   (name optional)
<itext> ::= (INNERITEXT | <invoke>)*
INNERITEXT =anything except TSTART, PIPE(s), TEND
```

Notice we can have nested invocation, so we could do something like `{{foo{bar}}|arg1{{ayy}}|..`
which will concatenate `bar` to `foo` and `ayy` to `arg1`

Now lets add parameters.

```
<program> ::= (OUTERTEXT |<invoke>)*
OUTERTEXT = anything, except for TSTART
<invoke> ::= (TSTART <itext> <targs> TEND) //TSTART/END are {{}}
    and itext is the name of the template
<targs> ::= (PIPE <itext>?)* //this creates a list of arguments
    (name optional)
<itext> ::= (INNERITEXT | <tvar>|<invoke>)* //tvar is
    {{{parameter}}}
INNERITEXT =anything except TSTART,VSTART, PIPE(s), TEND
<tvar> ::= VSTART VNAME (PIPE <itext>)? VEND //pipe <itext> says,
    if we don't find the definition for the parameter, we can
    specify a default
VNAME = anything, except PIPES, VEND
```

Finally, lets add definitions.

```
OUTERTEXT = anything, except for TSTART, or DSTART
INNERITEXT = anything, except for TSTART, DSTART, VSTART, PIPE(s),
    TEND
INNERDTEXT = anything, except for TSTART, DSTART, VSTART, PIPE(s),
    DEND
BODYTEXT = anything, except TSTART, DSTART, VSTART, DEND
VNAME = anything, except for PIPE(s), VEND

<program> ::= (OUTERTEXT | <invoke> | <define>)*
<invoke> ::= TSTART <itext> <targs> TEND
<targs> ::= (PIPE <itext>?)*
<itext> ::= (INNERITEXT | <tvar> | <invoke> | <define> )*
<tvar> ::= VSTART VNAME (PIPE <itext>)? VEND

<define> ::= DSTART <dtextn> <dparams> PIPES <dtextb> DEND
<dtextn> ::= (INNERDTEXT | <invoke> | <define> | <tvar>)*
<dparams> ::= (PIPE <dtextp>)*
<dtextp> ::= (INNERDTEXT | <invoke> | <define> | <tvar>)+
<dtextb> ::= (BODYTEXT | <invoke> | <define> | <tvar>)*
```

Note that a definition looks like: {`' foo | param1 | param2 || body'`} Note that we can have invocations, variables and definitions inside definitions, giving a lot of freedom. We an also have nothing in the name, allowing for anonymous functions. We MUST have non-empty parameter names however. Also note that the body can be empty.

What should {{{{{{ match as? We'll prioritize {{{ and make whitespace be significant in the language to fix the issue. But it will be trimmed in a lot of cases.

Notice that our language allows for users to write things that are syntactically correct, but will cause errors. For example, {`'foo||'`}{{ {{`foo`}}}} is syntactically correct, but the nested invoke will return null, and will try to invoke null, which is incorrect.

16.3 Evaluation of WML

We need to evaluate the AST. This takes in a tree, and spits out text.

For now, just imaging the text is one big string.

Say we have {{`foo|`{{`bar`}}`| args1`{{{`p1`}}}`|` {{`args2`}} `|` }}. To evaluate this invocation, we need to evaluate the name, then look up the name and find it in our environment. Now we need to evaluate the arguments.

When we see a definition, we need to evaluate each piece to figure out what they're called. We won't evaluate the body yet. (we only want to evaluate it when we invoke it). This will modify our environment.

16.4 Environment Model

We'll use static scoping. Our scopes have parents, names bound to text, etc.

To start, we have an outer environment E_0. It's parent is null.

Every evaluation takes an ASTNode, and an environment as a parameter. One concession, we will allow environments to be modified (not fully functional).

The program runs in the outer environment. So to eval the program, we need to do something like:

```
eval(OUTERTEXT,env) -> OUTERTEXT //any token is just itself
eval(list, env) -> if list==null -> "" //any list is done like this
             else -> let r = eval(head,env)
                  and then return r + eval(tail,env
```

For other things, just follow the recursive structure of the ASTNode.

```
eval(invoke(itext,args),env)->
   eval(itext,env)-> name
   eval(each arg,env)-> list of args //left to right
   look up name in env
      if(name found) make new child env E_1
      else look to parent
      if(am parent and not found)
         default.
   eval(body,E_1)
      return the string from the invoke
```

Executing the program is a matter of evaluating the entire AST, with a recursive descent. And at each step we will evaluate the node in the context of it's environment.

When we deal with parameters, there's an optional piece. `eval(tvar(name,optional),env)`, to deal with this, we have to look up the optional part. If we don't find it, we set it to a default value.

For definitions, we have: `eval(define(dtextn,dparams,dtextb),env)` When we recursively evaluate this:

```
eval(dtextn, env)

for each param in dparams
   eval(dparam,env)

//we dont evaluate the body!! we will do that when we actually
    call the function later!
```

```
create a binding
    name: list of params, body
    env: where this was defined.
add this binding to the env.
```

Note that defining returns an empty string. It simply has the side effect that the environment has been modified.

If we have something like: `{'foo || {{foo}}'}{{foo}}` This defines a function called foo, which takes no parameters, and whose body is the invoke of foo. We then invoke foo. What this is, is recursion. (Infinite at that)

In terms of environments, we have the outer env E_0 in which the function is defined.When it is invoked, we create a new environment E_1, who's parent is E_0. We then execute the body of E_1 which creates a new environment E_2 who's parent is E_1 and so on forever.

17 Computation and Conditionals

We need our language to be able to evaluate expressions and conditionals.

We could just build the whole sub-grammar for this, using a parser, we'll cheat and use a built-in Scala feature called reflection.

17.1 Evaluating expressions

```scala
import scala.reflect.runtime.universe
import scala.tools.reflect.ToolBox

val tb =
    universe.runtimeMirror(getClass.getClassLoader).mkToolBox()

tb.eval(tb.parse("3*4-2-1")).toString // returns "9"
tb.eval(tb.parse("3<=4")).toString //returns "true
```

This will parse a String into a value for us, then we turn it back to a String. It actually works for any valid Scala code. It will use the Scala compiler within that function!

17.2 Conditionals

These are structured like this: `{{#if|condition|then|else}}` where the else is optional. The condition will evaluate to the empty string for false, and non-empty for true.

We also have: `{{#ifeq | A | B | then | else}}`.

Note that all whitespace will be trimmed except for in body definitions.

17.2.1 Example Program with Problem: Factorial

```
{' fact | n || {{#ifeq | {{{n}}}|0|1
   | {{#expr | {{{n}}} * {{fact|{{#expr | {{{n}}}-1}}}}}}} '}
```

The problem with this is that there will be an infinite recursion. The issue is that the else also gets executed the way we have things set up. It will get thrown away at infinity, but we still end up computing it.

We need to delay the evaluation of the then/else parts until we know the value of the condition!

So far, other than if, we've been evaluating arguments as soon as we see them, before entering the body.

18 Lazy Evaluation

This is called **eager evaluation**. This uses call by value or call by reference. Recall the difference between Objects and values in Java.

In the if-statements, we need to do **lazy evaluation**. We don't evaluate the arguments at the call, we wait until we absolutely need to.

There are two forms of lazy evaluation in Scala.

Call-by-name: In Scala, you'd write `def foo(i => Int)` to perform a lazy evaluation. For example:

```scala
def ourConditional(condition:Boolean, thenPart:Unit,
    elsePart:Unit) = {
  if (condition){
    thenPart
  }else{
    elsePart
  }
}
```

Is the traditional way, which actually computes both sides as the parameters, whereas:

```scala
def ourConditional(condition: Boolean, thenPart => Unit, elsePart
    => Unit): Unit = {
  if(condition){
    thenPart
  }else{
    elsePart
  }
}
```

Only evaluates the parameters when they appear in the body. Note that even if we define a function inside the arguments of a function, their scope is still the outer scope. Also note that if you use the argument twice in the body, it will evaluate the parameter twice. (not good!)

`Call-by-need` avoids this issue, and only evaluates at most once.

This allows for things like Streams, which avoid stack-overflow errors in infinite or very large recursion, by lazily appending to the Stream.

```
def numsFrom(n:  Int):  Stream[Int] = n #::  numsFrom(n+1)
```

We can also declare values to be lazy:

```
def safeDivide(x:Int, y:Int) = {
  lazy val r = x/y
  if(y==0) "undefined"
  else r.toString
}
```

19 Closures

The problem right now is that eval takes a Node and Environment and returns a String. But how can we return functions?

To support the return a function, we need to be able to do both.

Consider this example:

```
{{  {'||foo'}}}
```

This would be kind of an ”anonymous” function. We're invoking a function with no name, which we can't currently do in this language.

Ideally, we want the evaluation to return either a String or a function.

But instead, we'll just return a pair, (String, Function).

Consider this example:

```
{{  {'|| foo'}{'||bar'}}}
```

What do we do? We'll just take the last one. Could be an error, but we won't include errors.

```
   {{x {'y||foo'}|{{y}}}}
```

This would we an invoke of a function called x, who's name includes the definition of another function y. This is fine, since definitions evaluate to the empty string. So the function we would invoke in this situation is just x. However, the definition still affected the environment, so when we refer to y as a parameter, we get it's body, foo

```
eval(program(list),env){
   if(list==Nil) return ("",null) //there is no program
   else if(list==head::tail)
      let h = eval(head,env) //evaluate the head
      let t = eval(tail,env) // recursively evaluate the tail.
      return(h._1 + t._1, if(t._2==null) h._2 else t._2) //recall
         we just take the last definition in a list of definitions
         so this is fine.

}
```

The final string is the concatenation of all the strings, and the last definition called.

```
   eval(invoke(itext,args),env){
      let name = eval(itext,env)
      evaluate the args..
      if(name._1.trim()=="") invoke name._2 //invoke the function!

      evaluate argument pairs...
}
```

```
eval (tvar(vname,itext),env){
   evaluate vname, turns into String
   lookup the vname in the environment

}
```

```
eval(define(dtextn,dparams,dtextb),env)
  let name = eval (dtextn,env) //returns a string (could be empty)
  if(name == "") return the function
  else add a binding to the environment.

  eval(dparams) //returns the string if empty, it's an error.

  return ("",function)
```

20 CPS

CPS stands for continuation passing style. A continuation is the code after a function call. When we call functions, we'll pass an extra parameter, which is the rest of the program. The functions will never actually return, instead it will just flow into the continuation (which the function has). It needs to be a lazy argument. Basically the function just invokes it's continuation instead of returning.

If the function has a return value, that value is simply passed as an argument into the continuation.

```
  def fact(n:Int): Int = {
    if (n==0)1
    else fact(n-1)*n
  }
  println(fact(5))
```

How would this work in CPS?

```
def fact(n:Int,c:(Int)=>Unit): Unit = {
if (n==0) c(1)
else fact(n-1,(z)=>{c(z*n)}) //creates new continuation z, (*n)
}
fact(5,(r) => {println(r)})
```

But y tho...

This has some performance issues... End up with serious stack problems.

But theres no need for a stack because we never return!

With CPS, we can actually return more than one value, since when we return we just pass another function with parameters, so these parameters can be whatever we want, and as numerous as we want.

We can use this as error control as well, one continuation for the correct control, one for the error.

21 Lambda Calculus

How minimal can we get a functional language that's still fully expressive?

Lambda calculus is one of the basic models of computation. The other is the Turing machine way. The idea is to represent computation entirely through functions.

The elements of the language that we'll have are:

- variables names (usually single letters)
- λ
- .
- brackets ()

The λ terms are defined recursively. λ terms are:

- variables
- If M and N are λ terms, then (MN) is also.
- If M is a λ term, and x is a variable, then $(\lambda x M)$ is a λ term.

Assume left associativity. And the body extends as far right as possible.

Application rule, (MN) represents calling a function. Where M is the function and N is the argument. The abstraction rule is for defining functions. λx.... where x is the name, and the stuff after the dot is the body.

The execution is a term-rewriting system. We turn some lambda term $M \to N$. This is called β-reduction. The $\to$ means "do some application". For example, $(\lambda x.x)y$ takes a new function x, who is just the identity,(returns x itself), and applies y. This is turns into y. We can view β-reduction in terms of functions and executions, but it's defined as just rewriting.

Example: $(\lambda x.M)N$ is executing M, but replacing all x's by N's.

Example: $(\lambda x.xx)y$ replaces all instances of x with y, so we end up with yy.

If we have a function like: $(\lambda x.x(\lambda x.x))y$ how does the scoping work? We only rewrite the outer x. So it turns into $y(\lambda x.x)$.

Works the way we expect as nested functions! But as a rewrite system, need to distinguish free from bound variables. A free variable is one that:

- x is free in: x
- if F is a set of free variables in the term M, then $F - \{x\} is free in (\lambda x.M)$
- if F is a set of free variables in the term M, and G is a set of free variables in N, then $F \cup G$ is free in (MN).

Anything that isn't free is bound.

Example: $(\lambda x.z(\lambda q.qz)(\lambda z.xq)x)$ here q is free in "q", z is free in "z", x is free in "x". (Base cases). $\{q\}$ and $\{z\}$ are free so $\{q, z\}$ are free in "qz". $\{q, z\}$ are free in "qz" so $\{z, q\} - \{q\} = \{z\}$ is free in $(\lambda q.qz)$. And so on

β-reduction formally uses this free/bound definition, but we understand it in terms of functions. (today). One thing we need to be careful of is the implicit brackets. (The ones we don't write based on order of operations).

Example: $(\lambda x.x(\lambda y.y)w)z$ we need to be careful, because what we actually have is: $(\lambda x.(x(\lambda y.y))w)z$ (remember left associativity) so we can only really reduce the outer redex, not the inner one. Recall that a redex MUST have a λ on the left, here we have an x on the left of the inner one. So we can rewrite as $(z(\lambda y.y)w)$. And there's nothing more we can do.

Counter Example: $(\lambda x.x((\lambda x.x)y))z$ here we've changed the order of operations using brackets.So now we actually have 2 redex's. So we can reduce to $(\lambda x.xy)z$, which reduces to zy. We reduced from inside out, but we could have gone outside in as well.

So we can apply β-reduction repeatedly and we get to a stuck point. But do we always reach a base case? (aka Normal Form). Do we always get to the same normal form? (Does the order of reduction matter?)

21.1 Church-Rossa Theorem

Given a λ-term M, if we do some reductions and reach a point N, and do some other reductions and reach P then $\exists Q$ such that N reduces to Q and P reduces to Q.

This is sometimes called the diamond property, or the confluence property. Basically this means order doesn't matter. Note that this theorem does NOT say anything about Q being the normal form. Doesn't say anything about whether or not there are infinite reductions. But it does say that if a normal form exists, it will be unique. (in terms of meaning, actual syntax might be different)

21.2 Non-Terminating λ Terms

It's pretty easy to come up with lambda terms that don't terminate. First some definitions:

$$\omega = (\lambda x.xx)$$
$$\Omega = (\omega\omega)$$

we can see that Ω reduces to Ω. Since we are replacing all instances of x in xx by $(\lambda x.xx)$ which gives $(\lambda x.xx)(\lambda x.xx) = \Omega$ so we can just continuously β-reduce this forever.

$$W = (\lambda x.xxx)$$

$$WW = (\lambda x.(xx)x)(\lambda x.(xx)x)$$

(brackets added for clarity)

$$= ((\lambda x.(xx)x)(\lambda x.(xx)x))(\lambda x.(xx)x)$$

$$= (WW)W$$
$$= ((WW)W)W$$

we can see that this keeps growing. But does the oder of reduction matter in terms of reaching the normal form *efficiently*?

Consider:

$$F = (\lambda x.(\lambda y.y))$$

note that this is syntactically equivalent to: $(\lambda xy.y)$

$$F(WW)I = II = I$$

since F just throws away it's input and returns the identity function. But that was from reducing the F first. We could've also reduced the WW first, which we know ends up in an infinite loop. However if we suddenly switch and reduce F, we'll get I, regardless of how many W's we accumulated in our "mistakes".

We get that doing the leftmost, outermost reduction first, guarantees finding a normal form, provided it exists.

21.3 Calculation with λ-Calculus

We can evaluate numbers as functions, called Church-numerals.

$$0 \equiv \lambda fx.x \equiv \lambda f.(\lambda x.x)$$
$$1 \equiv (\lambda fx.(fx))$$
$$2 \equiv (\lambda fx.f(fx))$$
$$3 \equiv (\lambda fx.f(f(fx))))$$