Lecture 5 Jan 25

```
scala> val s=1::2::3::Nil
s: List[Int] = List(1, 2, 3)

scala> s.map((x) = > x + 1)
<console>:1: error: ')' expected but '=' found.
s.map((x) = > x + 1)
          ^

scala> s.map((x) => (x + 1))
res0: List[Int] = List(2, 3, 4)

scala> s.map((x) => (x * x))
res1: List[Int] = List(1, 4, 9)

scala> val m = Map("Wallace" -> 0.3, "Esmeralda" -> 0.88)
m: scala.collection.immutable.Map[String,Double] = Map(Wallace -> 0.3, Esmeralda
 -> 0.88)
```

```
scala> m.map((x) => (x))
res3: scala.collection.immutable.Map[String,Double] = Map(Wallace -> 0.3, Esmeralda -> 0.88)

scala> m.map((x) => ((x._2,x._1)))
res4: scala.collection.immutable.Map[Double,String] = Map(0.3 -> Wallace, 0.88 -> Esmeralda)
```

```
scala> val s = 5
s: Int = 5

scala> s match {
     | case v: Int if(v>0) => "yes"
     | case _ => "no"
     | }
res5: String = yes
```

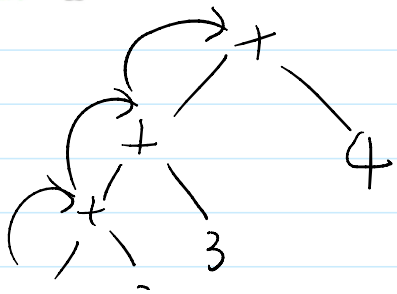
(a, b, c, d, e ···).map(f)

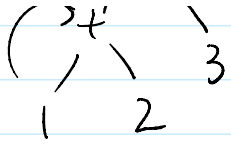
map ⇓ ⇓

f(a), f(b), f(c),

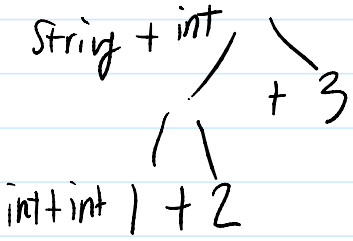
1, 2, 3, 4

```
scala> (1 to 5)
res6: scala.collection.immutable.Range.Inclusive = Range 1 to 5

scala> (1 to 5).reduceLeft(_ + _)
res7: Int = 15

scala> (1 to 5).reduceLeft((x,y) => { x + y } )
res8: Int = 15
```

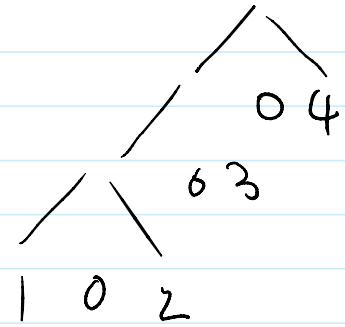




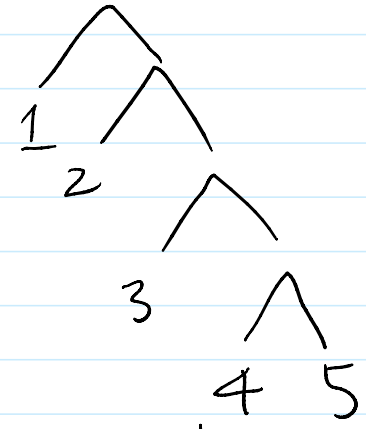
```
scala> (1 to 5).reduceLeft(_ * _)
res10: Int = 120

scala> (1 to 5).mkString
res11: String = 12345

scala> (1 to 5).mkString(",")
res12: String = 1,2,3,4,5
```



```
scala> (1 to 5).reduceLeft((s1,s2) => { s1 + "," + s2 })
<console>:12: error: type mismatch;
 found   : String
 required: Int
       (1 to 5).reduceLeft((s1,s2) => { s1 + "," + s2 })
                                              ^

scala> (1 to 5).reduceLeft((s1:Any,s2) => { s1 + "," + s2 })
res14: Any = 1,2,3,4,5

scala> (1 to 5).map(_.toString).reduceLeft((s1,s2) => { s1 + "," + s2 })
res15: String = 1,2,3,4,5
```



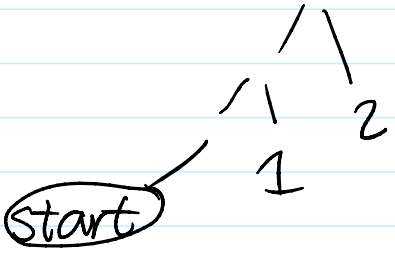
reduce-left



reduce-right

```
scala> ("!" ++ (1 to 5)).reduceLeft((s1:Any,s2) => { s1 + "," + s2 })
res17: Any = !,1,2,3,4,5

scala> (1 to 5).map(_.toString).foldLeft("!")((s1,s2) => s1 + "," + s2)
res18: String = !,1,2,3,4,5
```

foldLeft :



only accepts one argument and returns a function

```
scala> ("!" /: (1 to 5).map(_.toString))((s1,s2) => s1 + "," + s2)
res19: String = !,1,2,3,4,5
```

fun(x, y) => { . . . . }
can be viewed as a function takes only one argument and returns a function:

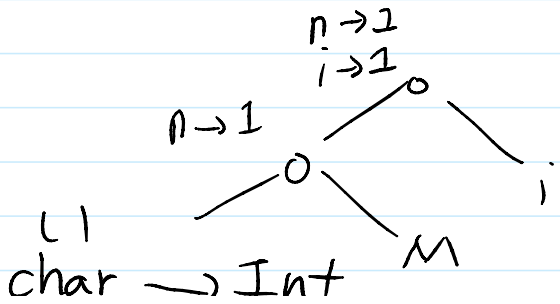
(fun(x))(y) => { . . . . }

```
scala> def prefix(p:String,s:String) = { p + s }
prefix: (p: String, s: String)String

scala> prefix("hello", " goooodbye")
res20: String = hello goooodbye

scala> val prefix = (p:String,s:String) => { p + s }
prefix: (String, String) => String = $$Lambda$1297/906440244@66604e6f

scala> prefix("hello", " goooodbye")
res21: String = hello goooodbye
```

```
scala> val prefix = (p:String) => { (s:String) => { p + s } }
prefix: String => (String => String) = $$Lambda$1308/250823362@183d22c5

scala> prefix("hello", " goooodbye")
<console>:13: error: too many arguments (2) for method apply: (v1: String)String => String in trait Func
tion1
       prefix("hello", " goooodbye")
                       ^

scala> prefix("hello")(" goooodbye")
res23: String = hello goooodbye

scala> prefix("hello")
res24: String => String = $$Lambda$1327/661442540@2fcc4d13
```

```
scala> def prefix (p:String)(s:String)  = { p + s }
prefix: (p: String)(s: String)String

scala> prefix("hello", " goooodbye")
<console>:13: error: too many arguments (2) for method prefix: (p: String)(s: String)String
       prefix("hello", " goooodbye")
                       ^

scala> prefix("hello")
<console>:13: error: missing argument list for method prefix
Unapplied methods are only converted to functions when a function type is expected.
You can make this conversion explicit by writing `prefix _` or `prefix(_)(_)` instead of `prefix`.
       prefix("hello")
              ^

scala> prefix("hello")(" goooodbye")
res27: String = hello goooodbye
```

map:



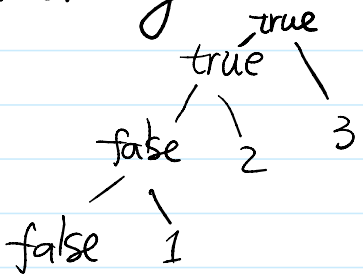
char → Int

```
scala> (Map[Char,Int]() /: "Mississippi") {
     | (m,c) => m + (c -> (m.getOrElse(c,0)+1))
     | }
res29: scala.collection.immutable.Map[Char,Int] = Map(M -> 1, i -> 4, s -> 4, p -> 2)
```

In general, list processing may need reductions or foldings, e.g. calculating something on the list.

Reduction doesn't need starting value.
Starting value or accumulating → folding.

Searching inside the list:



Use Boolean!

```
scala> val s = 1::2::3::4::5::6::Nil
s: List[Int] = List(1, 2, 3, 4, 5, 6)

scala> val t = 1::2::4::5::6::Nil
t: List[Int] = List(1, 2, 4, 5, 6)

scala> (false /: s)((b,i) => { if (!b && i==3) true else b })
res30: Boolean = true

scala> (false /: t)((b,i) => { if (!b && i==3) true else b })
res31: Boolean = false
```

if true, then carrying true all the way down.

var/val - declare local variables.
→ let x = ... in body
→ declare a local variable that lives in the body

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

redundancy

redundancy

```
val a = 1+xy
val b = 1-y
        x·a·a + y·b + a·b
```

```
scala> 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)
     | }
f: (x: Double, y: Double)Double
```

let - abstraction
→ nested, inner - function
→ parameter as "inner vars"

```
f ( ) {
    f'(a.) {
        f''( a, b ) }
```

```
    }
  }
}
```

Scope:
We refer the variables by names
→ variables hold values.

X → [3]

variables have lifetime.

```
function foo(a) {
    var i
    ⋮
}
```

i's lifetime

a's lifetime

}

- foo is also a value holding a function
- What is foo's lifetime?

Variable
→ declaration
↓
end of the function
global = end of everything

def

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

→ has to be alive at the same time.

→ Values also have lifetimes.

in C, void char *x = malloc();
↓ free(x)
// memories are alive until

→ Associate name : value in scopes

var x = 1	binding x : 1
⋮	⋮
x = 2	x : 2 rebinding x

Setting up bindings defines the program state