Lecture 3 Jan 18

Tuesday, February 6, 2018 9:55 PM

```
import scala.io.Source
val file = Source.fromFile("alice.txt", "UTF-8")
val text = file.mkString
val words = text.split("\\s+") // Array[String]
val lowers = for(w <- words if w(0).isLower) yield w
```

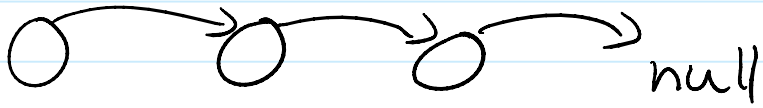
Tuples

..., triples, ... 5-tuples

```
scala> val t = ("balloon", 5, 2.0)
t : (String, Int, Double)

scala> t._1
res0 : String = balloon

scala> val(s,i,d) = t
s : String = baloon
i : Int = 5
d : Double = 2.0
```

```
d : Double = 2.0

scala> val grades = List("A", "A-")
grades : List[String] = List("A", "A-")
```

List



A::(B::(C::Nil))

```
scala> val grades = "A" :: "A-" :: "B+" :: Nil
grdes : List[String] = List("A", "A-", "B+")

scala> grades.head
rest1: String = A

scala> grades.tail
rest2: String = List("A-", "B+")

scala> grades.tail.head
rest3: String = A-
```

Functional Programming

→ functions are "first class citizens"
- We can pass functions as parameters, and return functions from functions as well.
- > Create dynamically

→ Immutability: we don't update data

→ Recursion: no for loops.

```
scala> def plus(a:Int, b:Int):Int = {a+b}
plus: (a:Int, b:Int)Int

// same as : a+b (without curly brackets)

scala> def incs (a:Int) = plus(a, 1)

scala> incs(5)
res3: Int = 6

scala> def prefixSum(n:Int):Int = {
         if (n==0) 0
         else prefixSum(n-1) +n
       }
```

```
        else  prefixSum(n-1) + n
    }
prefixSum: (n: Int) Int

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

fac(5)

→ fac(4) * 5

→ (fac(3) * 4) * 5

→ ((fac(2) * 3) * 4) * 5

→ (((fac(1) * 2) * 3) * 4) * 5

⋮

fac(0) → 1

1

2

6

24

120

→ We went all the way down to reach base case and came back.

foo() { : head recursion

base case and "came" back.

```
foo(){
   foo()
   compute()
}
```

head recursion → recursion first, then computation.

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

// m is accumulating as we go down
```

```
tailfac(3,1)
  → tailfac(2, 1*3)
     → tailfac(1, 1*3*2)
        → tailfac(0, 1*3*2*1)
```

Tail-Recursion:

```
foo(){
    compute()
    foo()
}
```

```
}
```

Advantages of tail recursion: Easy and efficient; optimize iteration

```
scala> def fac3(n:Int, m:Int = 1):Int = {
       if (n==0) m
       else fac3(n-1, n*m)
       }
```

```
scala> def fac3(n:Int): Int = {
         def helper(count:Int, prod:Int):Int = {
           if (count==0) prod
           else helper(count-1, prod * count)
         }
         helper(n, 1)
       }
```

```
def <name> (n:Int):Int = {
  if (n == someBase(ase) <baseResults>
  else <combine> (n, <name>(<reduce> (n)))
```

prefixSum		fac	
<name>	prefixSum	fac	
<combine>	+	*	
<base case>	0	0	
<baseresult>	0	1	
<reduce>	-1	-1	

```
scala> def iterPattern(n: Int, baseCase: Int, baseResult: Int,
                       combine: (Int, Int) => Int,
                       reduce: (Int) => Int): Int = {
         if (n == baseCase) baseResult
         else combine(n, iterPattern(                    ))
```