Functional Programming - Function Composition and Chai

Zeppelin

Functional Program...

Functional Programming - Function Composition and Chaining

more description here (https://en.wikipedia.org/wiki/Function_composition)

Notes from stackoverflow (http://stackoverflow.com/questions/2529184/difference-between-method-and-function-in-scala)

- Function has type, it defined by its arity and parameter type and return type; for example in Scala,

```
val addOne: (Int) => Int = (x) => x + 1
```

  We define a val (value), its name is `addOne` , which
  takes one parameter of type `Int` and return vaule of type `Int` , this is the type of this function. If we gives `Int` as parameter and returns a `Boolean` , it will be different type. This `value` is in the form of `function` .
  the function is implemented as described after `=` sign.
  In our example `x` is a parameter, can be any name,
  return `x + 1`

- A function can be invoked with a list of arguments to produce a result. A function has a parameter list, a body, and a result type.
- Functions that are members of a class, trait, or singleton object are called methods.
- Functions defined inside other functions are called local functions.
- Functions with the result type of Unit are called procedures.
- Anonymous functions in source code are called function literals. At run time, function literals are instantiated into objects called function values.
- Function can return a function; for example (https://www.youtube.com/watch?v=ugHslj60VfQ)

```
def deferTaxCalculation(emp: Employee): () => Double = {
reallySlowTaxCalculator(emp)
}
```

  deferTaxCalculator takes one `Employee` as parameter, then returns
  a function type `() = Double` , take no parameter and return a Double.

Input Data

```
ls: List[Int] = List(1, 2, 3, 4)
res30: List[Int] = List(1, 2, 3, 4)
```

Zeppelin

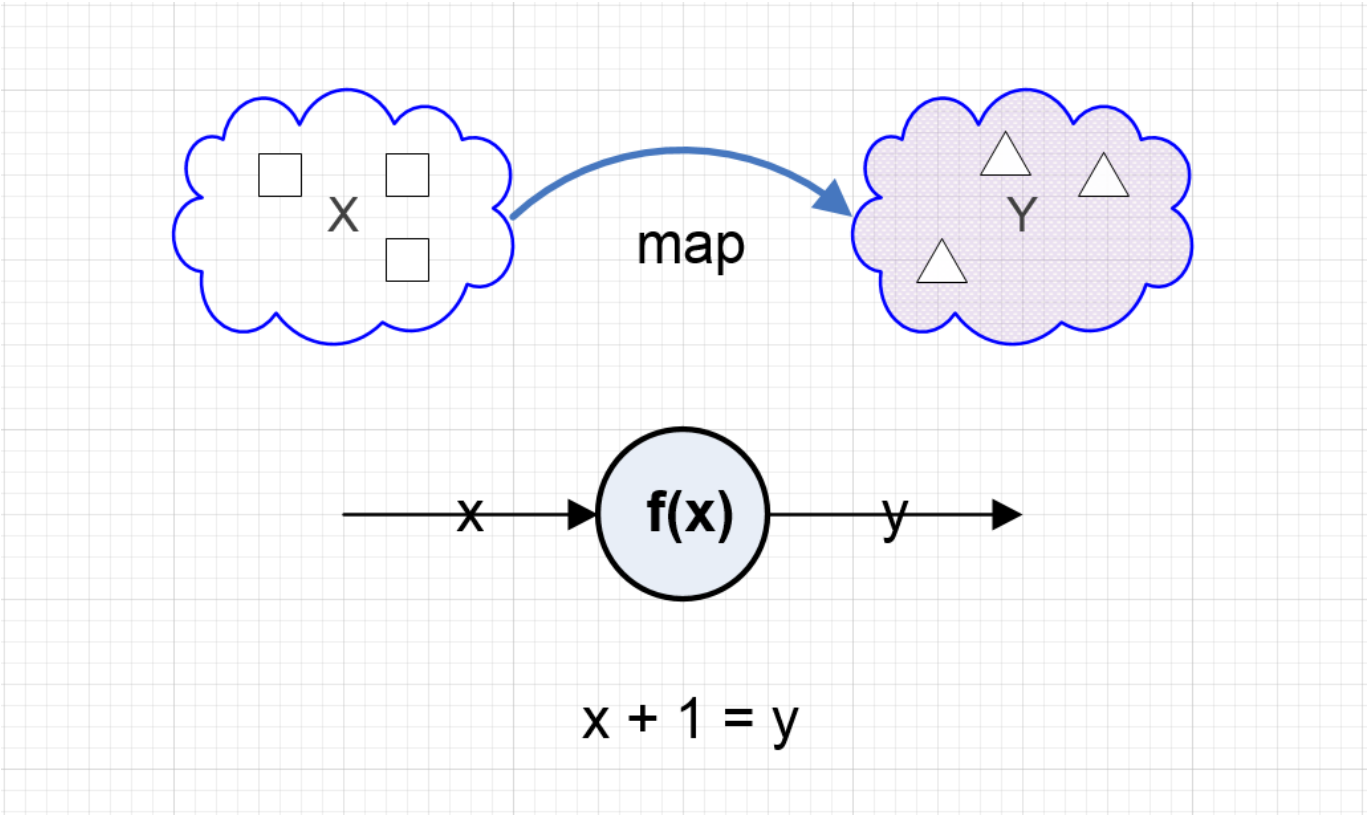
Function - example

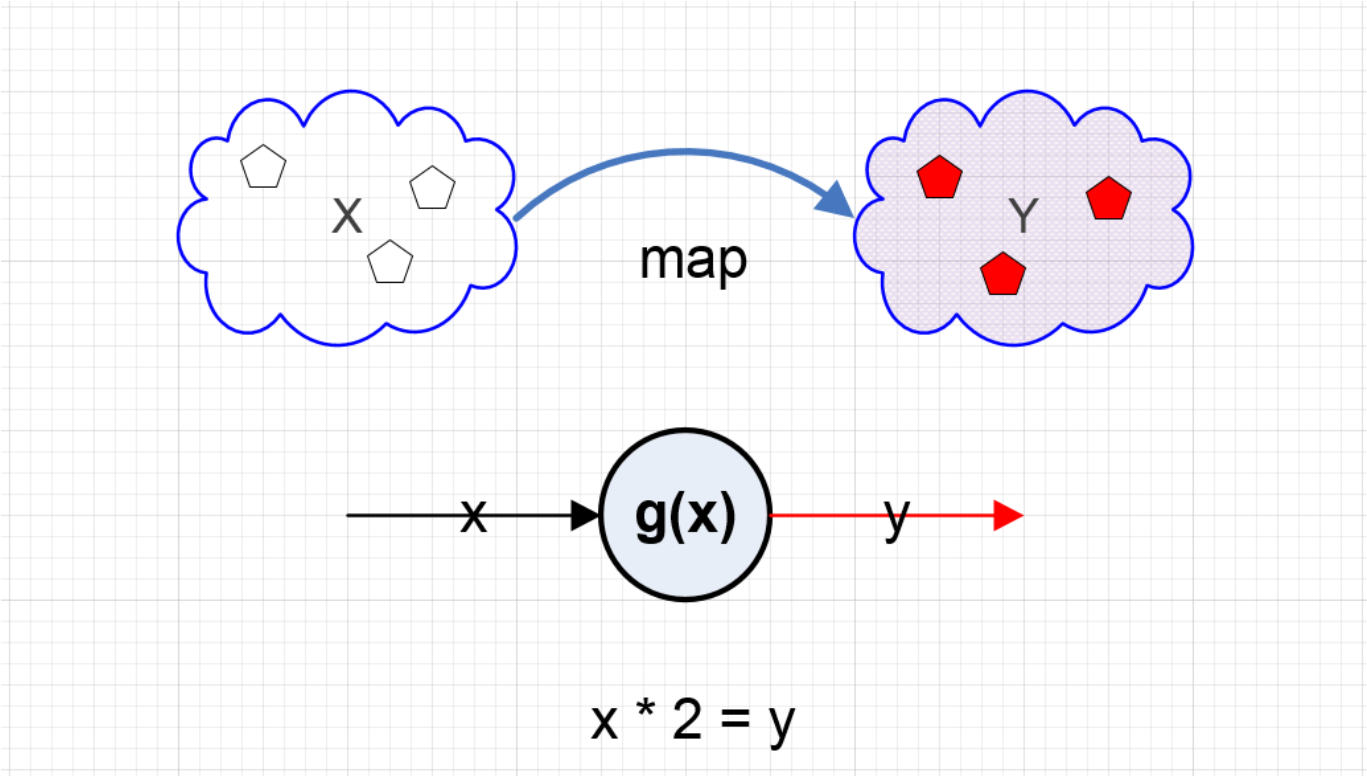
Functional Program...

```
f: (x: Int)Int
res31: Int = 3
```

Function - example

```
g: (x: Int)Int
res33: Int = 4
```



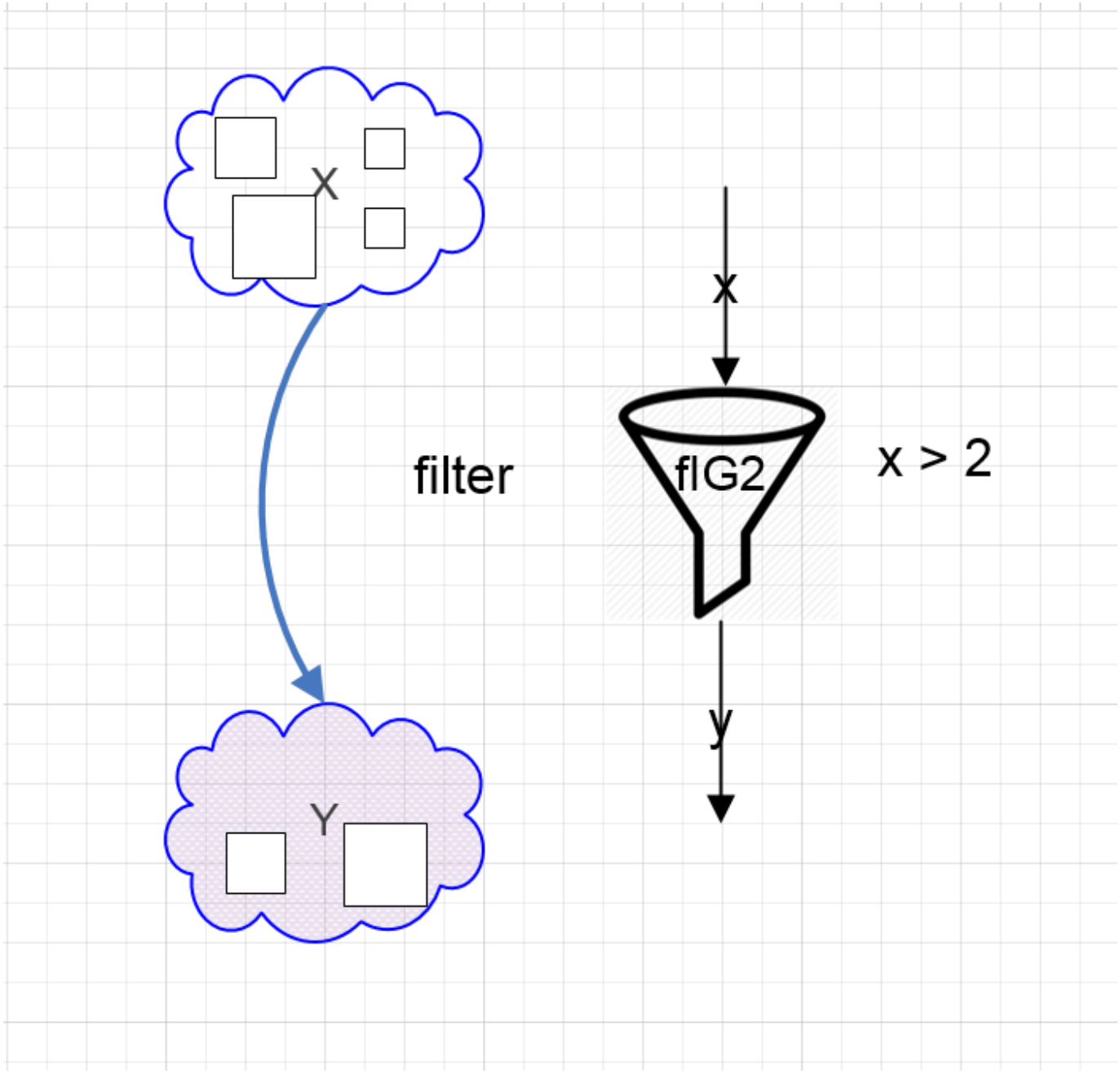


Method vs Function

```
m1: (x: Int)Int
res7: Int = 5
f1: Int => Int = <function1>
res8: Int = 5
f2: Int => Int = <function1>
res9: Int = 5
```

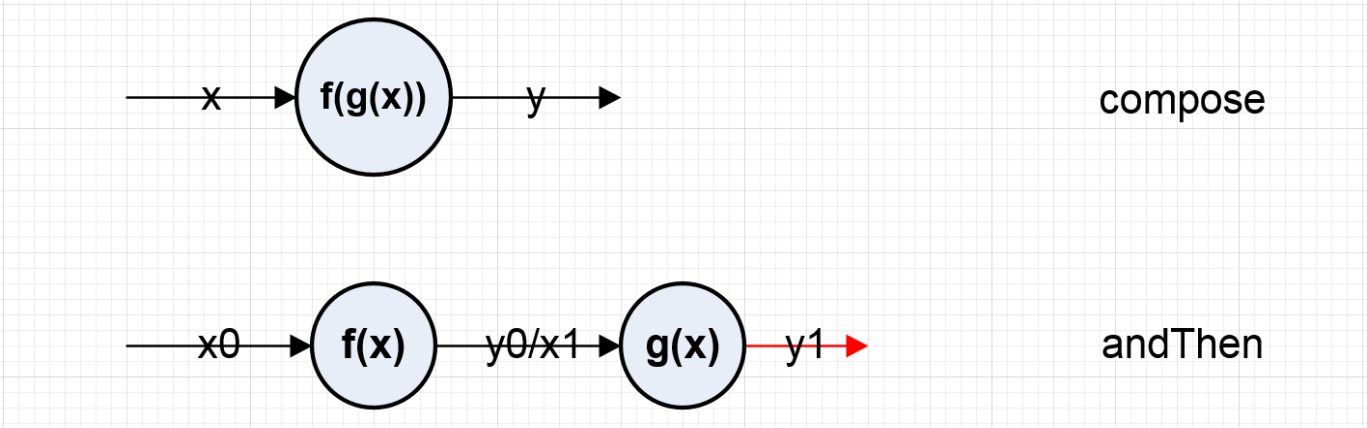
Filter

```
ls1: List[Int] = List(3, 4)
fIG2: Int => Boolean = <function1>
ls11: List[Int] = List(3, 4)
```



Compose and Chaining

```
ls2: List[Int] = List(3, 5, 7, 9)
ls3: List[Int] = List(4, 6, 8, 10)
```



Functional Chaining

```
fComposeG: Int => Int = <function1>
fAndThenG: Int => Int = <function1>
ls4: List[Int] = List(4, 6, 8, 10)
ls5: List[Int] = List(4, 6, 8, 10)
ls6: List[Int] = List(3, 5, 7, 9)
ls7: List[Int] = List(3, 5, 7, 9)
ls8: List[Int] = List(3, 5, 7, 9)
ls9: List[Int] = List(4, 6, 8, 10)
ls15: List[Int] = List(8, 10)


fn01: List[Int] => List[Int] = <function1>
ls16: List[Int] = List(8, 10)
ff: Int => Int = <function1>
fg: Int => Int = <function1>
ffComposeFg: Int => Int = <function1>
ls10: List[Int] = List(3, 5, 7, 9)
fncs: List[Int => Int] = List(<function1>, <function1>)
ls12: List[Int] = List(4, 6, 8, 10)
ls13: List[Int] = List(4, 6, 8, 10)
ls14: List[Int] = List(3, 5, 7, 9)
```

Chaining

Compare to

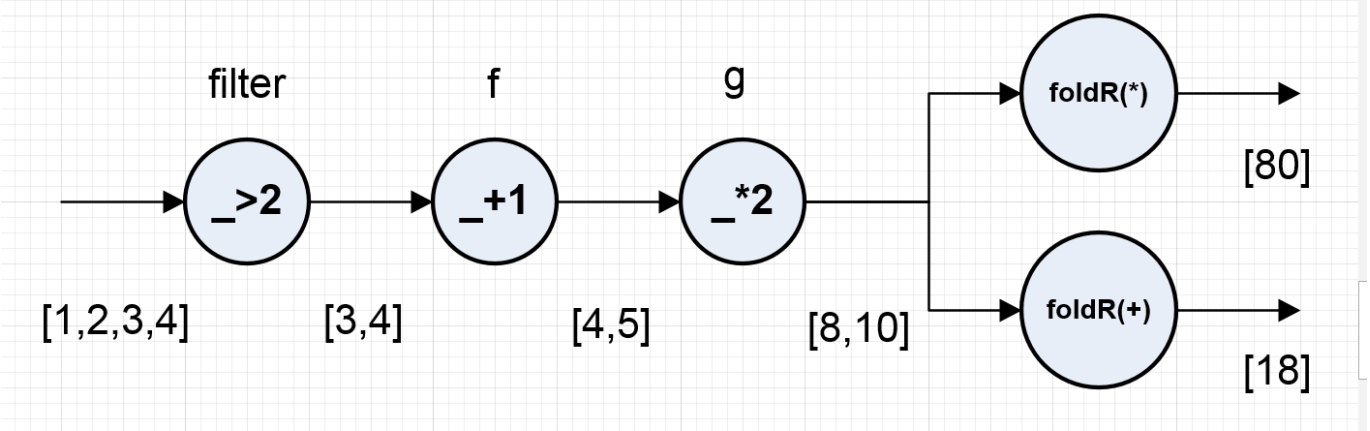
```
val ls15 = ls filter fIG2 map f map g     //3, 4 andThen + 1 andThen * 2
```

There are different ways to write chaining. We can define a function to represent a chain of functions and just call it with input, see example about `ls16` . we can treat `ls` as a parameter of a functions, which is a series of transformations.

```
//define a transform function
val fn01 = (xs: List[Int]) => {xs filter fIG2 map f map g}
val ls16 = fn01(ls)
```

There is a good discussion on stackoverflow (http://stackoverflow.com/questions/6137430/documenting-scala-functional-chains?rq=1). One comment in this discussion: “The best comments are the ones that explain why the code does something. Well-written code should make the “how” obvious from the code itself.”

```
res11: scala.collection.immutable.IndexedSeq[Int] = Vector(8, 10)
res12: Int => Boolean = <function1>
res13: Int => Int = <function1>
res14: Int => Int = <function1>
res15: Int => Int = <function1>
res16: scala.collection.immutable.IndexedSeq[Int] = Vector(8, 10)
res17: scala.collection.immutable.IndexedSeq[Int] = Vector(7, 9)
res18: scala.collection.immutable.IndexedSeq[Int] = Vector(7, 9)
```



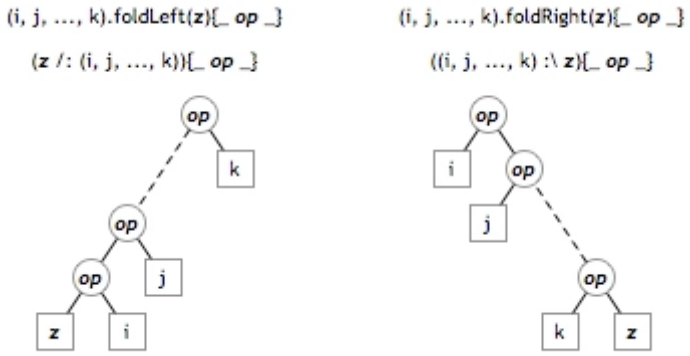
Fold vs For Loop

```
val xs = List(1, 2, 3, 4, 5)
var sum = 0
for (x <- xs) {
  sum = sum + x
}
println(sum) // => 15


xs.foldLeft(0) { (sum, x) =>
  sum + x
}
println(sum) //=> 15
```

from here (http://tototoshi.hatenablog.com/entry/20121223/1356197321)

foldLeft and foldRight

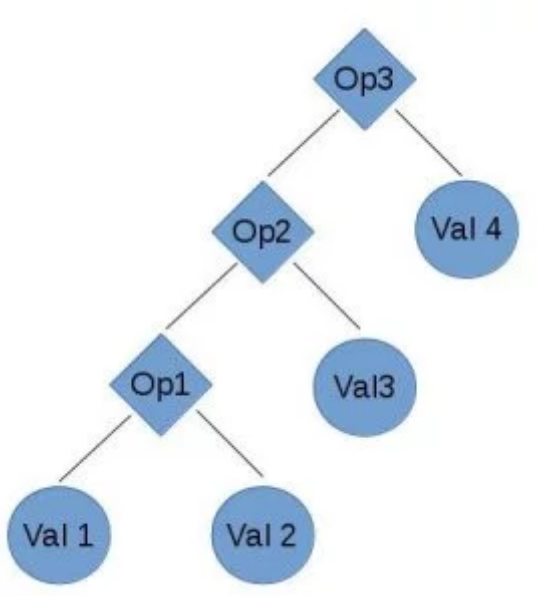


foldRight and foldLeft

```
res20: List[Int] = List(1, 2, 3, 4)
s4: Int = 10
s5: Int = 10
s6: Int = 24
s7: Int = 10
s8: Int = 80
s9: Int = 80
```

Reduce

If op for each node is different, then we can use `reduce` .



reduce(((val1 op1 val2) op2 val3) op3 val4)

```
res21: List[Int] = List(1, 2, 3, 4)
res22: Int = 10
res23: Int = 3
res24: Int = 2
op1: (Int, Int) => Int = <function2>
op2: (Int, Int) => Int = <function2>
res25: Int = 5
res26: Int = 18
```