Scala:
Higher-Order Function Deep Dive

By: Jack McDonald and Will Manning

Importance of higher-order functions

- In programming, it is sometimes necessary to abstract to a higher level
  - i.e. Writing a function that you can reference whenever to do some logic on something, rather than writing the same logic over and over
- Higher-order functions allow us to abstract to an even higher level
  - Functions are values, and therefore can be passed as parameters to higher-order functions

```
1  def sum(x: Int, y: Int): Int = {
2      x + y
3  }
4
5  def product(x: Int, y: Int): Int = {
6      x * y
7  }
8
9  def dividend(x: Int, y: Int): Int = {
10     x / y
11 }
12
13 def minus(x: Int, y: Int): Int = {
14     x - y
15 }
16
17 sum(2, 2) // : Int = 4
18 product(2, 2) // : Int = 4
19 dividend(2, 2) // : Int = 1
20 minus(2, 2) // : Int = 0
```

```
1  def atomicOps(x: Int, y: Int, f: (Int, Int) => Int) = {
2      f(x, y)
3  }
4
5  atomicOps(2, 2, (x, y) => x + y) //sum // : Int = 4
6  atomicOps(2, 2, (x, y) => x * y) //product // : Int = 4
7  atomicOps(2, 2, (x, y) => x / y) //dividend // : Int = 1
8  atomicOps(2, 2, (x, y) => x - y) //difference // : Int = 0
```

The two images represent the same computations; the difference being the introduction of a higher-order function allowing for code flexibility and abstraction. Now, when we want a function to operate on two integers we can simply use our higher-order function, atomicOps

foldLeft

- Can be thought of as a loop, iterating over a data structure from left to right
- **Parameters:** accumulator, data structure to recurse over, and a function f that defines ‘what to do’ as we loop over the data structure
- When folding over a tree, foldLeft is a great choice because unlike foldRight, it is tail-recursive and will prevent stack overflow

```
def myFoldLeft[A](acc: A)(t: Tree[A])(f: (A, A) => A): A = {
    def loop(acc: A)(t: Tree[A]): A = t match {
        case Empty[A]() => acc
        case Node(l, d, r) => {
            val lAndR = f(loop(acc)(l), loop(acc)(r))
            loop(f(d, lAndR))(Empty[A]())
        }
    }
    loop(acc)(t)
}
```

Map

- Higher-order function that can transform each element in a data structure
- **Parameters:** data structure and function f that defines the transformation for each element
- Can be thought of as a concise version of a for loop

```
def myMap[A, B](l: List[A])(f: A => B): List[B] = {
    def loop(l: List[A]): List[B] = l match {
        case Nil[A]() => Nil[B]()
        case Cons(h, tail) => Cons(f(h), loop(tail))
    }
    loop(l)
}
```

MapFirst

- Higher-order function that can transform the first element in a data structure
- **Parameters:** data structure, function f that checks if the element meets the condition, and function g that applies the transformation
- Can be thought of as a concise version of a for loop

```
def myMapFirst[A](l: List[A])(f: A => Option[A])(g: A => A): List[A] = l match {
    case Nil[A]() => l
    case Cons(h, t) => f(h) match {
      case None => {
            val tp = myMapFirst(t)(f)(g)
            Cons(h, tp)
        }
      case Some(hp) => Cons(g(hp), t)
    }
}
```

FlatMap

- Reduces inner-groupings of some sequence ‘flattens’
- Maps elements ‘map’
  - Is a mix between ‘flatten’ and ‘map’, hence, flatMap
- **Parameters:** List[A] to flatten (and map), callback function f

```
def myFlatMap[A, B](l: List[A])(f: A => List[B]): List[B] = {
    def loop(r: List[B])(l: List[A]): List[B] = l match {
        case Nil[A]() => if (r == Nil[B]()) Nil[B]() else concat(r, Nil[B]())
        case Cons(h, t) => loop(concat(r, f(h)))(t)
    }
    loop(Nil[B]())(l)
}
```

```
def concat[A](l1: List[A], l2: List[A]): List[A] = l1 match {
        case Cons(h, tail) => Cons(h, concat(tail, l2))
        case Nil() => l2 match {
            case Cons(h, tail) => Cons(h, concat(Nil(), tail))
            case Nil() => Nil()
        }
}
```