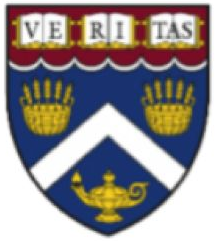
CSCI E-88A Introduction to Functional and Stream Processing for Big Data Systems

Harvard University Extension, Spring 2020
Marina Popova, Edward Sumitra



Lecture 06 - Scala Functional Programming (continued)

Agenda

- Admin info: Lab
- Recap of Functional Programming and Scala Introduction
- Functions
- List operations
- Maps
- Generic Programming

Functional Programming Recap

Tackle complexity in systems using

- Pure functions
- Immutable data
- Modularity through functional composition and higher-order-functions

“...we have to keep it crisp, disentangled, and simple if we refuse to be crushed by the complexities of our own making…” - Dijkstra

John Backus, Turing Award lecture, 1978(!)
A case for functional programming

https://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf

Introduction to Scala Programming Recap

Types, Values, and Expressions

Functions and Methods

Objects, classes, case classes and companion objects

List operations - map, filter, and reduce

```scala
case class Dog(name: String, age: Int) {
  def needsTraining: Boolean =
    if (age < 2) true else false

  def sayHello: String =
    s"${name} says woof!, woof!"

}

object Dog {
  def apply(csvRow: String): Dog = {
    val fields = csvRow.split( regex = ",")
    Dog(fields(0), fields(1).toInt)
  }
}
```

```scala
"create a dog from comma separated values" in {
  val dog1 = Dog("blackie,2")
  dog1.name should be("blackie")
  dog1.age should be (2)
}
```

Functions

- Defining functions as values
- Returning functions as output values
- Passing functions as input values

Higher Order Functions (HOF)
I.e, passing functions as inputs or returning functions as values

Defining Functions as values

Different ways of defining functions

- Define a method
- Function can be assigned to a value

```
def add2a(x: Int): Int = x + 2

val add2b: Int => Int = x => x + 2

val add2c: Int => Int = _ + 2
```

- Using shorthand for positional parameters

Functions

Different ways of defining functions

DEMO

Returning functions as values

Define a function to add 3 and a function to add 4

```
val add3a: Int => Int = x => x + 3

val add4a: Int => Int = x => x + 4
```

Is there a way to write the add N function generically?

(demo)

```
val addN: Int => Int => Int = n => x => x + n

val add3b: Int => Int = addN(3)

val add4b: Int => Int = addN(4)
```

Functions as Input Values

Generate data points to plot a chart

```
val generateXY: (Int => Int) => List[(Int, Int)] = { f =>
  List(0,10,20,30,40,50)
    .map(x => (x, f(x)))
}

def generateAB(f: Int => Int): List[(Int, Int)] = {
  List(0,10,20,30,40,50)
    .map(x => (x, f(x)))
}
```

Demo

Higher Order Functions

```
def andThen[A](g: (R) => A): (T1) => A
```

Composes two instances of Function1 in a new Function1, with this function applied first.

A	the result type of function g
g	a function R => A
returns	a new function f such that `f(x) == g(apply(x))`
Annotations	@unspecialized()

```
def compose[A](g: (A) => T1): (A) => R
```

Composes two instances of Function1 in a new Function1, with this function applied last.

A	the type to which function g can be applied
g	a function A => T1
returns	a new function f such that `f(x) == apply(g(x))`
Annotations	@unspecialized()

From https://www.scala-lang.org/api/current/scala/Function1.html

Demo

List operations

Map, filter, reduce

Take:

Drop:

Grouped:

GroupBy:

Zip:

Scan:

```
scala> List(1,2,3,4,5).take(3)
res30: List[Int] = List(1, 2, 3)

scala> List(1,2,3,4,5).drop(2)
res31: List[Int] = List(3, 4, 5)

scala> List(1,2,3,4,5).grouped(2).toList
res32: List[List[Int]] = List(List(1, 2), List(3, 4), List(5))

scala> List(1,2,3,4,5).groupBy(_ % 2)
res33: scala.collection.immutable.Map[Int,List[Int]] = Map(1 -> List(1, 3, 5), 0
 -> List(2, 4))

scala> List(1,2,3,4,5).zip(List(10,20,30,40,50))
res34: List[(Int, Int)] = List((1,10), (2,20), (3,30), (4,40), (5,50))

scala> List(1,2,3,4,5).scan(0)(_ + _)
res35: List[Int] = List(0, 1, 3, 6, 10, 15)
```

See https://www.scala-lang.org/api/current/scala/collection/immutable/List.html

List operations

Demo

List flatMap

Flatmap works on functions that produce sequences

```
final def flatMap[B](f: (A) => IterableOnce[B]): List[B]
```

Builds a new list by applying a function to all elements of this list and using the elements of the resulting collections.

E.g., function that produces a sequence

```
def add2: Int => Int = _ + 2

def uptoN: Int => List[Int] = n => (0 to n).toList
```

```
scala> uptoN(3)
res39: List[Int] = List(0, 1, 2, 3)

scala> List(1,2,3).map(uptoN)
res40: List[List[Int]] = List(List(0, 1), List(0, 1, 2), List(0, 1, 2, 3))

scala> List(1,2,3).flatMap(uptoN)
res41: List[Int] = List(0, 1, 0, 1, 2, 0, 1, 2, 3)
```

List flatMap

Flatmap for nested sequences

```
def allPairs_a: Seq[(Int, Int)] =
  (1 to 5).flatMap { a =>
    (10 to 50 by 10).map {b =>
      (a,b)
    }
  }
```

Demo

for comprehension

Shorthand syntax for flatMap and map

```
def allPairs_b: Seq[(Int, Int)] =
  for {
    a <- (1 to 5)
    b <- (10 to 50 by 10)
  } yield (a,b)
```

Demo

for comprehension

Right angle triangle sides

```
def rightTriangleTriples: Seq[(Int, Int, Int)] =
  for {
    a <- 1 to 10
    b <- 1 to 10
    c <- 1 to 10
    if (a * a + b * b == c * c)
  } yield (a,b,c)
```

Demo

Maps

Maps are associations of keys to values
Type: Map[K,V], K = type of key, V = type of value
Properties are mostly the same as Java maps
Maps are immutable by default!

Creating Maps

```
scala> Map(1 -> "A", 2 -> "B")
res48: scala.collection.immutable.Map[Int,String] = Map(1 -> A, 2 -> B)

scala> Map((1, "A"), (2,"B"))
res49: scala.collection.immutable.Map[Int,String] = Map(1 -> A, 2 -> B)

scala> List((1, "A"), (2,"B")).toMap
res50: scala.collection.immutable.Map[Int,String] = Map(1 -> A, 2 -> B)

scala>

scala> Map.empty[Int,String]
res51: scala.collection.immutable.Map[Int,String] = Map()
```

Maps

Map methods

https://www.scala-lang.org/api/current/scala/collection/immutable/Map.html

Accessing Map elements

```
scala> val myMap = Map(1 -> "A", 2 -> "B")
myMap: scala.collection.immutable.Map[Int,String] = Map(1 -> A, 2 -> B)

scala> myMap(2)
res52: String = B

scala> myMap.get(2)
res53: Option[String] = Some(B)
```

Accessing Map elements for non-existent keys

```
scala> myMap(3)
java.util.NoSuchElementException: key not found: 3
  at scala.collection.immutable.Map$Map2.apply(Map.scala:138)
  ... 36 elided

scala> myMap.get(3)
res60: Option[String] = None

scala> myMap.getOrElse(3,"C: by default")
res61: String = C: by default
```

Maps

Map key-value:, Map.map(f) where
f: (K,V) => W

Map values: Map.transform(f) where
f: (K,V) => W

Demo

Generic Programming

Functions and Classes can take types as a parameter.
Code modularity through reuse
Particularly useful for container classes like List and Map

Generic functions

```
def initializeWith[A](initVal: A, n: Int): List[A] =
  List.fill(n)(initVal)
```

Type Variance

From wikipedia:
Variance refers to how subtyping between complex types relates to subtyping between their components.
Focus on container types

Covariant: +A
If A and B are two types and A is a subtype of B, then Container[A] is a subtype of Container[B]
E.g., List[Cat] is a subtype of List[Animal]

Contravariant: -A
If A and B are two types and A is a subtype of B, then Container[B] is a subtype of Container[A]

Invariant; A
Neither covariant or contravariant, but fixed to the declared type.

Type Variance examples

Containers:

```
sealed abstract class  List[+A]
```

```
trait  Map[K, +V]
```

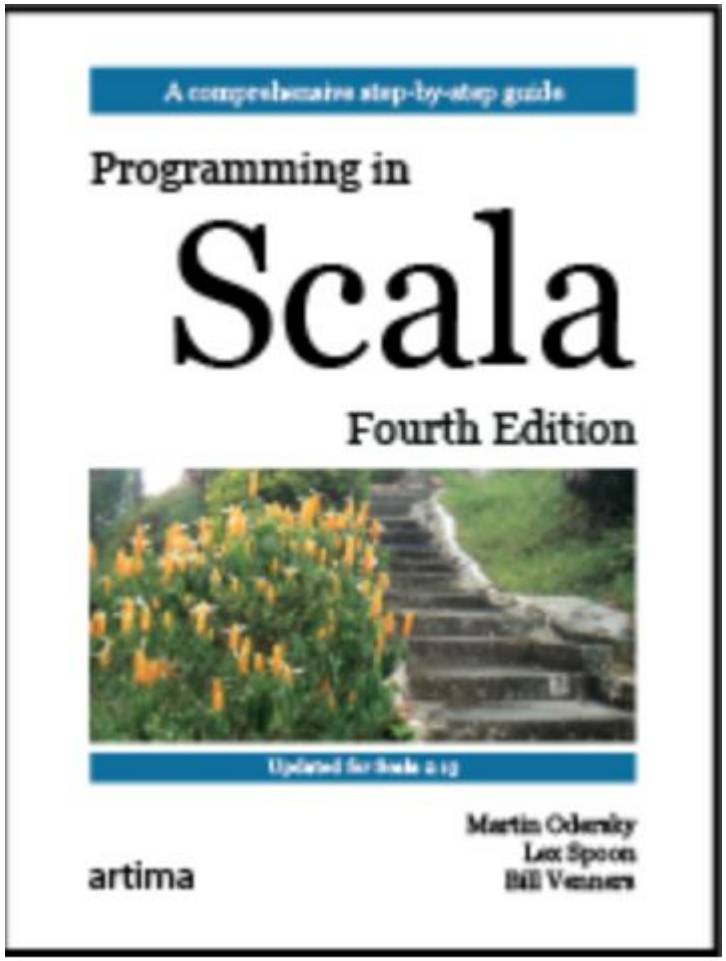
Functions:

```
trait  Function1[-T1, +R]
```

Scala Reference

Programming in Scala

[Programming in Scala, First Edition]



@Marina Popova, Edward Sumitra