A taste of functional programming

Highlights from the very beginning of *Sergei Winitzki, The Science of Functional Programming: A tutorial, with examples in Scala*

András Simon

December 10, 2025

"Functional programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input."

—Functional Programming HOWTO

$1 * 2 * \ldots * 10 = 3628800$

```
scala> (1 to 10).product
val res67: Int = 3628800
```

$$f(n) = \prod_{i=1}^{n} i$$

$$f(10) = 3628800$$

```
def f(n: Int) = (1 to n).product

scala> f(10)
res6: Int = 3628800
```

Nameless function

$n \mapsto \prod_{i=1}^{n} i$

```
(n: Int) => (1 to n).product
```

Example (not typical – see the rest of the slides for typical ones):

```
scala> ((n: Int) => (1 to n).product)(10)
val res7: Int = 3628800
```

Transforming sequences

$$double(a) = \langle 2a_i : i \in \mathrm{dom}(a) \rangle$$

```
def double(a: List[Int]) = a.map(x => 2*x)

scala> double(List(3,2,4))
val res16: List[Int] = List(6, 4, 8)

scala> double((1 to 8).toList)
val res19: List[Int] = List(2, 4, 6, 8, 10, 12, 14, 16)

scala> List(3,2,4).sortBy(x => -x)
val res58: List[Int] = List(4, 3, 2)
```

The argument of `sortBy` is a function that computes the sorting key from a sequence element.

Aggregating sequences

```
scala> val lst = (5 to 10).toList.map(x => x%3)
val lst: List[Int] = List(2, 0, 1, 2, 0, 1)

scala> lst.max
val res29: Int = 2

scala> lst.size
val res30: Int = 6

scala> lst.sum
val res66: Int = 6
```

Aggregating sequences

$$countEven(L) = \sum_{n \in L} isEven(n) \qquad \text{where}$$

$$isEven(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ 0 & \text{otherwise.} \end{cases}$$

```
def isEven(n: Int) : Int = 1 - n%2
def countEven (lst:List[Int]) = lst.map(isEven).sum

scala> countEven(List(3,2,4))
val res17: Int = 2
```

Or we could just use a nameless function:

```
def countEven(lst: List[Int]): Int = lst.map(x => 1 - x%2).sum
```

Aggregating sequences

Test for primality: $P(n) \iff 1 < n \ \& \ (\forall k \in [2, n-1]) n \bmod k \neq 0$

```
def P(n: Int) = 1 < n & (2 to n - 1).forall(k => n%k != 0)
\\ or
def P(n: Int) : Boolean =
  1 < n & (2 to n - 1).forall(k => n%k != 0)

scala> P(13)
res4: Boolean = true

scala> P(14)
res5: Boolean = false

// a variant (! is negation):
def P(n: Int) = 1 < n & !(2 to n - 1).exists(k => n%k == 0)
```

Filtering sequences

Filtering and truncating a sequence

```
scala> val lst = (1 to 10).toList
val lst: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> lst.filter(x => x%3 == 0)
val res31: List[Int] = List(3, 6, 9)

scala> lst.takeWhile(x => x%3 != 0)
val res33: List[Int] = List(1, 2)
```

Examples

- $$avg(a) = \frac{1}{|\text{dom}(a)|} \sum_{i \in \text{dom}(a)} a_i$$

```
def avg(a: List[Double]) : Double = a.sum / a.size
def avg(a: List[Double]): Double

scala> avg(List(1.0, 2.0, 3.0))
val res88: Double = 2.0
```

- $$\sum_{k \in [1,10], \cos k > 0} \sqrt{\cos k} = ?$$

```
scala> (1 to 10).
| filter(k => math.cos(k) > 0).
| map(k => math.sqrt(math.cos(k))).
| sum
val res87: Double = 3.115810288505257
```

Examples

- Define a function add20 of type `List[List[Int]] => List[List[Int]]` that adds 20 to every element of every inner list.

```
def add20(a : List[List[Int]]) : List[List[Int]] =
   a.map( lst => lst.map (x => x+20))

scala> add20( List( List(1), List(2, 3) ) )
val res106: List[List[Int]] = List(List(21), List(22, 23))
```

- Define a function that takes a list a of integers and a function $f : \mathbb{Z} \to \mathbb{Z}$ as arguments, and returns $\max\{f(a_i) : i \in \mathrm{dom}(a)\}$.

```
def maxf(a: List[Int], f: Int => Int) : Int = a.map(f).max

scala> maxf(List(2, 3, 4, 5), x => 60 / x)
val res89: Int = 30
```

(With a one-argument function like `f` above, we can shorten `.map(x => f(x))` to `.map(f)`.)

Tuples

```
scala> val tup = (1, "one", 0.5)
val tup: (Int, String, Double) = (1,one,0.5)

scala> tup._2
val res36: String = one

// destructuring:
scala> val (x,y,z) = tup
val x: Int = 1
val y: String = one
val z: Double = 0.5

scala> y
val res37: String = one
```

Tuples

```
scala> val tup = ((1, "one"), 0.5)
val tup: ((Int, String), Double) = ((1,one),0.5)

scala> val (x,y,z) = tup // -> ERROR

scala> val ((x,y),z) = tup
val x: Int = 1
val y: String = one
val z: Double = 0.5
```

Pattern matching

```
scala> tup
val res39: ((Int, String), Double) = ((1,one),0.5)

scala> tup match { case ((a, b), c) => a + c }
val res40: Double = 1.5

scala> val basket: List[(String, (Int, Double))] =
     | List(("apples", (3, 0.5)),
     | ("pears", (2, 0.8)), ("lemons", (0, 1.0)))
val basket: List[(String, (Int, Double))] =
  List((apples,(3,0.5)), (pears,(2,0.8)), (lemons,(0,1.0)))

scala> basket.map({case (fruit, (count, unitPrice))
     | => (fruit, count * unitPrice)})
val res70: List[(String, Double)] =
  List((apples,1.5), (pears,1.6), (lemons,0.0))
```

Maps (a.k.a. dictionaries)

```
scala> basket
val res76: List[(String, (Int, Double))] =
  List((apples,(3,0.5)), (pears,(2,0.8)), (lemons,(0,1.0)))

scala> basket.map({case (fruit, (count, unitPrice))
     | => (fruit, unitPrice) })
val res78: List[(String, Double)] =
  List((apples,0.5), (pears,0.8), (lemons,1.0))

scala> val basketMap =
     | basket.map({case (fruit, (count, unitPrice))
     | => (fruit, unitPrice) }).toMap
val basketMap: Map[String, Double] =
  Map(apples -> 0.5, pears -> 0.8, lemons -> 1.0)

scala> basketMap.apply("pears")
val res80: Double = 0.8
```

Maps (a.k.a. dictionaries)

```
scala> basketMap.groupBy( (fruit, unitPrice)
     | => if (unitPrice < 0.6 ) "cheap" else "expensive" )
val res81: Map[String, Map[String, Double]] = HashMap(
  cheap -> Map(apples -> 0.5),
  expensive -> Map(pears -> 0.8, lemons -> 1.0))
```

Examples

- Given a list of integers, how many of them are bigger than the next one? (The method `tail` returns the list w/o its first element.)

```
def biggerCount(lst: List[Int]) : Int =
   lst.zip(lst.tail).filter( (x,y) => x > y ).size

scala> biggerCount(List(1,3,2,4,5,6))
val res61: Int = 1

scala> biggerCount(List(1,3,2,4,5,6,1))
val res62: Int = 2
```

The method `count` takes a predicate and returns the number of sequence elements for which the predicate is true; using it we can get rid of creating a new list just to take its length:

```
def biggerCount(lst: List[Int]) : Int =
   lst.zip(lst.tail).count( (x,y) => x > y )
```

Examples

- Define a function that takes a list *a* of integers and a function $f : \mathbb{Z} \to \mathbb{Z}$ as arguments, and returns the first element of *a* with the biggest *f*-value. (Cf. the earlier `maxf`!)

```
scala> def maxf2(a: List[Int], f: Int => Int) =
     | a.zip(a.map(f)).maxBy( (x, y) => y )._1
def maxf2(a: List[Int], f: Int => Int): Int

scala> maxf2(List(2, 3, 4, 5), x => 60 / x)
val res96: Int = 2
```

`maxBy` is like `max`, except that that it applies the one-argument function it is given as argument to produce the values to be compared. So `.max` is `.maxBy( x => x )`.

Examples

- Find all integers $k \in [1, 10]$ such that at least three different integers $0 \leq j \leq k$ satisfy the condition $j^2 > 2k$.

```
scala> (1 to 10).
     | filter( k => (1 to k).
     | filter( j => j*j > 2*k).
     | size >= 3)
val res103: IndexedSeq[Int] = Vector(6, 7, 8, 9, 10)
```

Examples

- Given a `List[(String, Int)]` of purchases (`(fruit, amount)` pairs) compute a Map[String, Int] showing the total counts. So, for the input:

```
Seq(("apple", 2), ("pear", 3), ("apple", 5),
  ("lemon", 2), ("apple", 3))
```

the output must be:

```
Map("apple" -> 10, "pear" -> 3, "lemon" -> 2)
```

Examples

```
scala> val purchases = List(("apple", 2), ("pear", 3),
  ("apple", 5), ("lemon", 2), ("apple", 3))
val purchases: List[(String, Int)] = List((apple,2),
  (pear,3), (apple,5), (lemon,2), (apple,3))

scala> purchases.groupBy( (fruit, amount) => fruit)
val res114: Map[String, List[(String, Int)]] =
 HashMap(apple -> List((apple,2), (apple,5), (apple,3)),
 pear -> List((pear,3)), lemon -> List((lemon,2)))
```

Examples

```
scala> purchases.groupBy( (fruit, amount) => fruit).
     | map( (fruit, dict) => (fruit, dict.
     | map((_, amount) => amount)))
val res115: Map[String, List[Int]] =
 HashMap(apple -> List(2, 5, 3),
 pear -> List(3), lemon -> List(2))

scala> purchases.groupBy( (fruit, amount) => fruit).
     | map( (fruit, dict) => (fruit, dict.
     | map((_, amount) => amount).sum))
val res119: Map[String, Int] = HashMap(apple -> 10,
  pear -> 3, lemon -> 2)
```