Functional Programming in Scala
Part IV
Other Collections
(Vector, Array, String, Set, Map)

Raj Sunderraman

Vector

A more balanced sequence; Faster random access to elements.
Very similar to lists otherwise.

val nums = Vector(1,2,3,4)
val people = Vector(“Bob”,”James”,”Peter”)

Vectors support same operations as lists except for ::
Instead of x :: xs, there is

x +: xs - creates a new vector with lead element x followed by all elements of xs

xs :+ x - creates a new vector with trailing element x preceded by all elements of xs

Class Hierarchy:

— Iterable
  — Seq
    — String, Array, List, Vector, Range
  — Set
  — Map

Arrays, Strings, Ranges

Arrays and Strings support same operations as Seq and can be implicitly converted into Sequences whenever needed.

```
val xs : Array[Int] = Array(1,2,3)
xs map (x => 2 * x)

val ys: String = “Hello World”
ys filter (_.isUpper)
```

Ranges are sequences of evenly spaced integers (to, until, by keywords)

```
val r: Range = 1 until 5  // 1,2,3,4
val s: Range = 1 to 5     // 1,2,3,4,5
1 to 10 by 3              // 1,4,7,10
6 to 1 by -2              // 6,4,2
```

Some more Sequence Operations

xs exists p	true if there is an element x in xs such that p(x) is true
xs forall p	true if p(x) is true for all x in xs
xs zip ys	A sequence of pairs drawn from corresponding elements of xs and ys
xs.unzip	reverse of zip
xs.flatMap f	Applies a collection generating function to each element of xs and returns a flat list by concatenating the results
xs.sum	sum of all elements of xs
xs.product	product of all elements of xs
xs.max	max element of xs
xs.min	min element of xs

<u>examples:</u>

(1 to 5) flatMap (x => (1 to 5) map (y => (x,y)))
will give us Vector((1,1),(1,2),…,(5,5)), i.e. all combinations

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
    (xs zip ys).map(pair => pair._1 * pair._2).sum

def isPrime(n: Int): Boolean = (2 until n) forall (d => n%d!= 0)
```

Handling Nested Sequences - Example

Given a positive integer, n, find all pairs of positive integers i and j with 1 <= j < i < n such that i + j is prime.

For example, if n = 7, the sought pairs will be
(2,1), (3,2), (4,1), (4,3), (5,2), (6,1), (6,5)

— Generate all pairs such that 1 <= j < i < n
— Filter the pairs for which i + j is prime.

Generate Pairs:

```
((1 until n) map (i => (1 until i) map (j => (i, j)))).flatten
```

Using the law: xs flatMap f = (xs map f).flatten

```
(1 until n) flatMap (i => (1 until i) map (j => (i, j)))
```

Filter:

```
(1 until n) flatMap (i => (1 until i) map (j => (i, j)))
  filter (pair => isPrime(pair._1 + pair._2))
```

For Comprehensions

Higher order functions such as map, flatMap, or filter provide powerful constructs to manipulate lists.

But sometimes these expressions become hard to understand. For example, the previous problem.

Scala’s for-comprehensions come to the rescue!

Example:

```
class Student(n: String, a: Int) {
  var name: String = n;
  var age: Int = a;
}

val s1 = new Student("Jones",25)
val s2 = new Student("Smith",35)
var students = List(s1,s2)

for (s <- students if s.age > 30) yield s.name

res3: List[String] = List(Smith)
```

For Comprehensions Syntax

for (s) yield e

where s is a sequence of generators and filters and e is an expression whose value is returned by an iteration.

A generator is of the form p <- e, where p is a pattern and e an expression whose value is a collection.

A filter is of the form “if f”, where f is a boolean expression

The sequence must start with a generator.

instead of (s), we may write {s} if writing the for in multiple lines.

For Comprehensions Examples

Given a positive integer, n, find all pairs of positive integers i and j with 1 <= j < i < n such that i + j is prime.

```
for {
    i <- 1 until n
    j <- 1 until i
    if isPrime(i+j)
  } yield (i,j)
```

Scalar Product

```
def scalarProduct(xs: List[Double], ys: List[Double]): Double =
  (for ((x,y) <- (xs zip ys)) yield x * y).sum
```

Sets

Sets are another basic abstraction in the Scala collection.

```
val fruit = Set(“apple”,”banana”,”pear”)
val s = (1 to 6).toSet
```

Most operations on sequences are also available on sets.

```
s.map(_ + 2)
fruit filter(_.startsWith == “app”)
s.nonEmpty
```

Main differences between sets and sequences:

1. Sets are unordered
2. Sets do not have duplicate elements
3. Fundamental operation on sets is “contains”
   e.g. (s contains 5)

n Queens problem

Given a n x n chess board, place n queens so that none of them are attacked by any other queen.

- Recursively solve for (k-1) queens
- Place kth queen such that it is not attacked by any of the previous queens

```
type Queen = (Int,Int)
type Solutions = List[List[Queen]]
```

For example, if n=4, and lets say we have placed 3 queens already:

```
queens = List((1,0),(2,3),(3,1))
```

and we have to place the 4th queen. The choices will be

```
(0,0), (0,1), (0,2), (0,3)
```

For each, we have to verify it it is attacked by previous queens.

n Queens solution

```
type Queen = (Int, Int)
type Solutions = List[List[Queen]]

def queens(n: Int) = {
  def attacked(q1: Queen, q2: Queen) =
    ((q1._1 == q2._1) || (q1._2 == q2._2) ||
    ((q1._1-q2._1).abs == (q1._2-q2._2).abs))

  def isSafe(queen: Queen, others: List[Queen]): Boolean =
    others forall (x => !attacked(queen, x))

  def placeQueens(k: Int): Solutions = {
    if (k == 0) List(Nil)
    else
      for {
        queens <- placeQueens(k-1)
        col <- 0 until n
        if isSafe((k-1, col), queens)
      } yield  (k-1, col) :: queens
  }

  placeQueens(n)
}
```

Scala Maps (Dictionary)

A Map consists of pairs of keys-values (also called mappings/associations)

key -> value
and
(key, value)

are treated the same.

val states1 = Map(“AL” -> “Alabama”, “AK” -> “Alaska”)
creates an immutable Map

var states2 = scala.collection.mutable. Map(“AL” -> “Alabama”, “AK” -> “Alaska”)
creates a mutable Map

states2 += (“AZ”->”Arizona”, “CO”->”Colorado”)
states -= “AL”
states -= (“AZ”,”CO”)
states(“AK”) = “Alabama!”

Scala Maps (Dictionary)

Lookups:

`ms get k`
`The value associated with key k in map ms as an option, None if not found.`

`ms(k)`
(or, written out, `ms apply k`) The value associated with key `k` in map `ms`, or exception if not found.

`ms getOrElse (k, d)`
The value associated with key `k` in map `ms`, or the default value `d` if not found.

`ms contains k`
Tests whether `ms` contains a mapping for key `k`.

`ms isDefinedAt k`
Same as `contains`.

Scala Maps (Dictionary)

Additions and Updates:

`ms + (k -> v)`
The map containing all mappings of `ms` as well as the mapping `k -> v` from key `k` to value `v`.

`ms + (k -> v, l -> w)`
The map containing all mappings of `ms` as well as the given key/value pairs.

`ms ++ kvs`
The map containing all mappings of `ms` as well as all key/value pairs of `kvs`.

`ms updated (k, v)`
Same as `ms + (k -> v)`.

Removals:

`ms - k`
The map containing all mappings of `ms` except for any mapping of key `k`.

`ms - (k, l, m)`
The map containing all mappings of `ms` except for any mapping with the given keys.

`ms -- ks`
The map containing all mappings of `ms` except for any mapping with a key in `ks`.

Scala Maps (Dictionary)

Subcollections:

`ms.keys`
An iterable containing each key in `ms`.

`ms.keySet`
A set containing each key in `ms`.

`ms.keysIterator`
An iterator yielding each key in `ms`.

`ms.values`
An iterable containing each value associated with a key in `ms`.

`ms.valuesIterator`
An iterator yielding each value associated with a key in `ms`.

Transformation:

`ms filterKeys p`
A map view containing only those mappings in `ms` where the key satisfies predicate `p`.

`ms mapValues f`
A map view resulting from applying function `f` to each value associated with a key in `ms`.

Scala Maps - Frequency Count

Given a text file, produce a frequency count of all characters in the file.

e.g. file a.txt contains

Upsets defined the NCAA tournament for most of the last two weeks, marking even more madness this March than usual. Sister Jean became the most famous nun in sports. A No. 16 seed (UMBC) topped a No. 1 seed (Virginia) for the first time in the men's tournament. Buffalo busted brackets. So did Kansas State and Florida State and Syracuse.

```
val source = scala.io.Source.fromFile("a.txt")
val lines = try source.mkString finally source.close()
val s = for (c <- lines) yield c.toUpper
val m = Map[Char,Int]()
val freq = s.foldLeft(Map[Char,Int]() withDefaultValue 0) ((m, c) => m updated (c, m(c)+1))

res0: scala.collection.immutable.Map[Char,Int] =
Map(E -> 32,
-> 1, . -> 7, N -> 22, T -> 27, Y -> 1, J -> 1, U -> 11, F -> 9, A -> 26,
M -> 13, ) -> 2, I -> 13,   -> 59, ’ -> 1, , -> 1, G -> 2, 6 -> 1, 1 -> 2,
V -> 2, L -> 4, B -> 5, P -> 4, C -> 6, H -> 8, W -> 2, ( -> 2, K -> 4,
R -> 14, O -> 17, D -> 12, S -> 28)
```