

# Fundamentos de Programación Funcional y Concurrente

## Colecciones

Juan Francisco Díaz Frias

Profesor Titular (1993-hoy)  
[juanfco.diaz@correounivalle.edu.co](mailto:juanfco.diaz@correounivalle.edu.co)  
Edif. B13 - 4009



**Universidad del Valle**

Septiembre 2023

# Plan

## 1 Secuencias

- Vectores, cadenas y rangos
- Expresiones *For* y Búsqueda Combinatoria

## 2 Conjuntos

- Expresiones *For*
- Conjuntos

## 3 Expresiones *For*

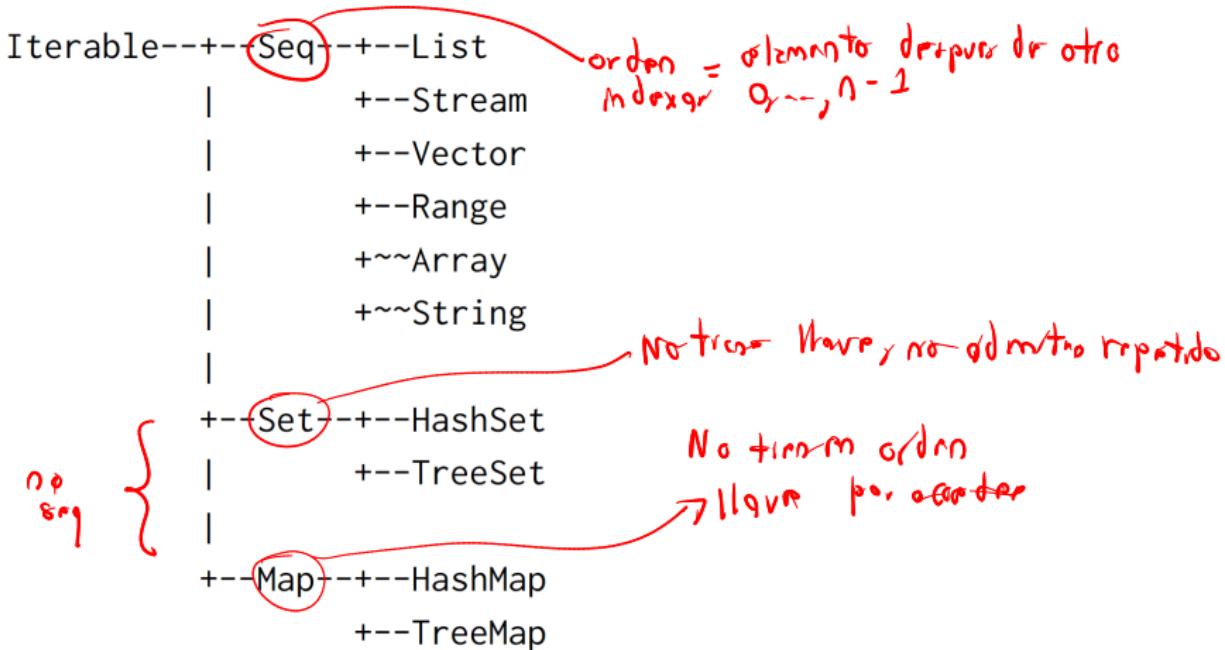
- Consultas con *For*
- Traducción del *For*
- Generalización del *For*

## 4 Maps o Asociaciones

## 5 Streams o Flujos

- Flujos
- Evaluación perezosa
- Calculando con secuencias infinitas

# Las colecciones en Scala



# Contents

## 1 Secuencias

- Vectores, cadenas y rangos
- Expresiones *For* y Búsqueda Combinatoria

## 2 Conjuntos

- Expresiones *For*
- Conjuntos

## 3 Expresiones *For*

- Consultas con *For*
- Traducción del *For*
- Generalización del *For*

## 4 Maps o Asociaciones

## 5 Streams o Flujos

- Flujos
- Evaluación perezosa
- Calculando con secuencias infinitas

# Vectores

- Las **listas** son secuencias *lineales*: el acceso al primer elemento es mucho más rápido que el acceso a los del medio o a los últimos de la lista.
- Scala define una implementación alternativa de las secuencias: **Vector**. Provee un acceso más balanceado en tiempo a los elementos de la secuencia.
- Los vectores se crean de forma análoga a las listas:

```
0  val nums = Vector(10, 12, 6, 8)
1  val personas = Vector("Juan", "Pedro", "Mariana")
```

- Los vectores soportan las mismas operaciones que las listas, salvo ::. En su lugar hay dos operadores binarios + : y : +:
  - x : + xs: Crea un nuevo vector cuyo primer elemento es x, seguido por los elementos de xs
  - xs : + x: Crea un nuevo vector cuyo último elemento es x, antecedido por los elementos de xs
- *List* y *Vector* son subclases de *Seq* que es subclase de *Iterable*.

# Rangos

- Otro tipo sencillo de secuencias son los **rangos**.
- Un rango representa una secuencia de enteros espaciados uniformemente.
- Hay tres operadores para crear rangos: **to** (inclusivo), **until** (exclusivo) y **by** (para determinar el paso uniforme). Por ejemplo:

```
0 scala> val r:Range = 1 to 5
1 val r: Range = Range 1 to 5
2
3 scala> val s:Range = 1 until 5
4 val s: Range = Range 1 until 5
5
6 scala> 1 to 10 by 3
7 val res3: scala.collection.immutable.Range = Range 1 to 10 by 3
8
9 scala> 6 to 1 by -2
10 val res4: scala.collection.immutable.Range = inexact Range 6 to 1 by -2
```

- Los rangos se representan internamente como objetos con tres campos: cota inferior, cota superior y paso.

# Arreglos y Cadenas

- Los arreglos (*Array*) y cadenas (*String*) soportan las mismas operaciones que *Sq* y pueden ser convertidos implícitamente a secuencias cuando sea necesario. (No son subclases de *Sq* porque provienen de Java)
- Por ejemplo:

```
0  scala> val xs:Array[Int] = Array (1,2,3)
1  val xs: Array[Int] = Array(1, 2, 3)
2
3  scala> xs map (x => 2*x)
4  val res14: Array[Int] = Array(2, 4, 6)
5
6  scala> val ys:String = "Hola-mundo"
7  val ys: String = Hola mundo
8
9  scala> ys.filter(_.isUpper)
10 val res16: String = H
```

# Operaciones sobre secuencias

- $xs \ exists \ p$ : *true* si hay un elemento  $x$  de  $xs$  tal que  $p(x)$ . *false* sino
- $xs \ forall \ p$ : *true* si todo elemento  $x$  de  $xs$  cumple  $p(x)$ . *false* sino
- $xs \ zip \ ys$ : devuelve una secuencia de parejas de elementos de  $xs$  con elementos de  $ys$  en la misma posición.
- $xs.unzip$ : si  $xs$  es una secuencia de parejas, devuelve una pareja con las listas de los primeros y los segundos elementos de la pareja.
- $xs.flatMap \ f$ : devuelve la secuencia resultante de aplicar  $f$  a cada elemento de  $xs$ .
- $xs.sum$ : devuelve la suma de todos los elementos de  $xs$  si esta es una secuencia numérica
- $xs.prod$ : devuelve el producto de todos los elementos de  $xs$  si esta es una secuencia numérica
- $xs.max$ : devuelve el máximo de todos los elementos de  $xs$  si esta es una secuencia de elementos que tiene asociada un ordenamiento
- $xs.min$ : devuelve el mínimo de todos los elementos de  $xs$  si esta es una secuencia de elementos que tiene asociada un ordenamiento

# Ejemplos

- Listar todas las parejas de números enteros  $x$  y  $y$  tal que  $x$  está entre 1 y  $M$  y  $y$  está entre 1 y  $N$

```
0 (1 to M) flatMap (x=> (1 to N) map (y=> (x,y)))
```

¿Por qué se usa *flatMap* en lugar de *map*?

- Calcular el producto escalar de dos vectores:

```
0 def prodEscalar(xs:Vector[Double], ys:Vector[Double]):Double =
1   (xs zip ys).map(xy=>(xy._1*xy._2)).sum
2 prodEscalar(Vector(0.5, 1, 1.5), Vector(1.5, 2, 2.5))
```

- Calcular el producto escalar de dos vectores (versión con función con reconocimiento de patrones):

```
0 def prodEscalar2(xs:Vector[Double], ys:Vector[Double]):Double =
1   (xs zip ys).map({case (x,y)=>x*y }).sum
```

# Ejercicio

Un número  $n$  es **primo** si sólo es divisible por 1 y por  $n$ . Y  $n$  es **compuesto** si no es primo.

- Escriba un test de primalidad de muy alto nivel. En este caso, privilegie que sea conciso sobre que sea eficiente.

```
0 def esPrimo(n:Int):Boolean = ???
```

- Escriba un test de composición de muy alto nivel. En este caso, privilegie que sea conciso sobre que sea eficiente.

```
0 def esCompuesto(n:Int):Boolean = ???
```

# Manejo de secuencias anidadas

- Se puede extender el uso de funciones de alto orden sobre secuencias para expresar cálculos que son típicamente expresados usando ciclos anidados.
- **Ejemplo:** Dado un entero positivo  $n$ , genere todas las parejas de enteros positivos  $(i, j)$  tal que  $1 \leq j < i < n$  y  $i + j$  es primo. Por ejemplo si  $n = 7$ , las parejas serían

  
 $\{(2, 1), (3, 2), (4, 1), (4, 3), (5, 2), (6, 1), (6, 5)\}$

- Una forma sencilla de lograrlo es:
  - Generar la secuencia de parejas  $(i, j) : 1 \leq j < i < n$ 
    - Generar la secuencia de enteros entre 2 y  $n$  (excluido)
    - Para cada  $i$ , generar la secuencia de parejas  $(i, 1), (i, 2), \dots, (i, i - 1)$
  - Filtrar de esa secuencia los pares para los que  $i + j$  es primo

# Implementación en Scala

- Generar la secuencia de parejas  $(i, j) : 1 \leq j < i < n$ 
  - Generar la secuencia de enteros entre 2 y  $n$  (excluido)

```
0  scala> (2 until 7)
1  val res26: scala.collection.immutable.Range = Range 2 until 7
```

- Para cada  $i$ , generar la secuencia de parejas  $(i, 1), (i, 2), \dots, (i, i - 1)$

```
0  scala> (2 until 7).flatMap(x=>(1 until x).map(y=>(x,y)))
1  val res23: IndexedSeq[(Int, Int)] = Vector((2,1), (3,1), (3,2), (4,1), (4,2), (4,3),
2  (5,1), (5,2), (5,3), (5,4), (6,1), (6,2), (6,3), (6,4), (6,5))
```

¿Por qué se usa *flatMap*?

- Filtrar de esa secuencia los pares para los que  $i + j$  es primo

```
0  scala> (2 until 7).flatMap(x=>(1 until x).map(y=>(x,y)))
1      filter(pareja => esPrimo(pareja._1 + pareja._2))
2  val res25: IndexedSeq[(Int, Int)] = Vector((2,1), (3,2), (4,1), (4,3), (5,2), (6,1), (6,5))
```

La solución es muy elegante, pero ¿se puede escribir más sencillo?

# Contents

## 1 Secuencias

- Vectores, cadenas y rangos
- Expresiones *For* y Búsqueda Combinatoria

## 2 Conjuntos

- Expresiones *For*
- Conjuntos

## 3 Expresiones *For*

- Consultas con *For*
- Traducción del *For*
- Generalización del *For*

## 4 Maps o Asociaciones

## 5 Streams o Flujos

- Flujos
- Evaluación perezosa
- Calculando con secuencias infinitas

# Expresiones For

- Las funciones de alto orden como *map*, *flatMap* y *filter* proveen construcciones muy **poderosas y expresivas** para manipular secuencias.
- Pero, sucede con frecuencia, que **el nivel de abstracción es tan alto** que la expresión se vuelve difícil de entender.
- Scala provee las **expresiones For** para mantener la expresividad facilitando el entendimiento de la semántica de lo que se escribe.

## Sintaxis de una expresión For

Una expresión *For* es de la forma:

for i → 1 to 10  
yield 1

0    `for ( s ) yield e`

donde *s* es una secuencia de **generadores** y **filtros**, y *e* es una expresión cuyo valor es retornado en cada iteración.

- Un **generador** es una expresión de la forma  $p < - e$  donde *p* es un patrón y *e* es una expresión que se evalúa a un valor de tipo Colección.
- Un **filtro** es una expresión de la forma *if f* donde *f* es una expresión booleana.
- La secuencia *s* debe comenzar por un generador.
- Si hay varios generadores en la secuencia, el último varía más rápido que el primero.

En lugar de `( s )`, se pueden usar corchetes `{ s }`. En ese caso, la secuencia de generadores y filtros se puede escribir en varias líneas, sin tener que usar `";"`.

## Ejemplo de expresión For

Suponga que *personas* es una secuencia de elementos de la clase *Persona*, con campos *nombre* y *edad*:

```
0 case class Persona(nombre: String, edad: Int)
```

Para obtener los nombres de las personas mayores de 20 años, se puede escribir:

```
0 for (p <- personas if p.edad > 20) yield p.nombre
```

lo cual es equivalente a escribir:

```
0 personas filter (p => p.edad > 20) map (p => p.nombre)
```

Las **expresiones For** son similares a los ciclos en lenguajes imperativos, salvo que, además, construye una lista con los resultados de cada iteración.

# Uso de expresiones For

- Dado un entero positivo  $n$ , genere todas las parejas de enteros positivos  $(i, j)$  tal que  $1 \leq j < i < n$  y  $i + j$  es primo.

```
0 for {
1   i <- 2 until 7
2   j <- 1 until i
3   if esPrimo(i+j)
4 } yield (i,j)
```

- Hallar el producto escalar de dos vectores:

```
0 def prodEscalar(xs:Vector[Double], ys:Vector[Double]):Double =
1   (for ((x,y) <- xs zip ys) yield (x*y)) .sum
```

# Conjuntos en Scala

- Los conjuntos (*Set*) son otras abstracción básica de las colecciones de Scala.
- Los conjuntos se escriben similarmente a las secuencias:

```
0 val frutas=Set("lulo", "banana", "curuba", "chontaduro")
1 val s = (1 to 6).toSet
```

- La mayoría de las operaciones sobre secuencias, están disponibles sobre conjuntos

```
0 s map (_ + 2)
1 frutas filter (_.startsWith == "c")
2 s.nonEmpty
```

# Conjuntos vs Secuencias

Las principales diferencias entre conjuntos y secuencias son:

- Los conjuntos no tienen orden. Entonces las operaciones que tengan que ver con el orden no aplican a los conjuntos.
- Los conjuntos no tienen elementos duplicados.

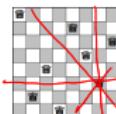
```
0  scala> val s = (1 to 6).toSet
1  val s: scala.collection.immutable.Set[Int] = HashSet(5, 1, 6, 2, 3, 4)
2
3  scala> s map (_ / 2)
4  val res36: scala.collection.immutable.Set[Int] = HashSet(0, 1, 2, 3)
```

- Una operación fundamental sobre los conjuntos es *contains*:

```
0  scala> s contains 5
1  val res38: Boolean = true
```



# Ejemplo: las $N$ reinas



- Considere el problema de colocar  $N$  reinas en un tablero de ajedrez de  $N \times N$ , de manera que no se ataquen entre ninguna de ellas.
- Nótese que puede haber diversas maneras de colocar las reinas y que no se ataquen. Cada una de ellas es válida. Luego resolver el problema, en este caso, significa calcular el **conjunto** de soluciones válidas. Cada elemento del conjunto, es una ubicación de las  $N$  reinas en donde no se atacan entre ellas.
- Una manera de resolver el problema consiste en colocar una reina en cada fila, fila por fila, empezando con el tablero vacío.
- **Idea:** Una vez se han colocado  $k - 1$  reinas, en las primeras  $k - 1$  filas, se cuenta con el conjunto de soluciones hasta ese momento. Ahora se toma cada una de esas soluciones, y se trata de colocar la  $k$ -ésima reina en la fila  $k$  en una columna donde no se ataque con ninguna otra reina. Pueden existir varias opciones que funcionen. Se recolectan todas las opciones.

# La idea implementada en Scala

```

0 def reinas(n:Int)={  

1   def ataques(col:Int, dist:Int, reinas>List[Int]):Boolean = {  

2     // devuelve true si la reina colocada en la columna col  

3     // de la fila reinas.length + dist  

4     // ataca alguna reina colocada en reinas  

5     ...  

6   }  

7   def esSeguro(col:Int, reinas>List[Int]):Boolean = {  

8     // devuelve true si colocar la reina de la fila  

9     // reinas.length +1 en la columna col, sabiendo que  

10    // las otras reinas estan ubicadas segun la lista reinas  

11    // es seguro. Devuelve false en caso contrario  

12    ...  

13  }  

14  

15  def coloqueReinasHastaLaFila(k:Int):Set[List[Int]] = {  

16    if (k==0) Set(List())  

17    else  

18      for {  

19        reinas <- coloqueReinasHastaLaFila(k-1)  

20        col <- 0 until n  

21        if esSeguro(col, reinas)  

22      } yield col::reinas  

23  }  

24  coloqueReinasHastaLaFila(n)  

25 }

```

# Las funciones que faltan...

```

0  def reinas(n:Int)={  

1    def ataques(col:Int, dist:Int, reinas>List[Int]):Boolean = {  

2      // devuelve true si la reina colocada en la columna col  

3      // de la fila reinas.length + dist  

4      // ataca alguna reina colocada en reinas  

5      reinas match {  

6        case Nil => false  

7        case r::otrasReinas =>  

8          col == r ||(col - r).abs == dist ||  

9            ataques(col, dist +1, otrasReinas)  

10       }  

11     }  

12     def esSeguro(col:Int, reinas>List[Int]):Boolean = {  

13       // devuelve true si colocar la reina de la fila  

14       // reinas.length +1 en la columna col, sabiendo que  

15       // las otras reinas estan ubicadas segun la lista reinas  

16       // es seguro. Devuelve false en caso contrario  

17       !ataques(col,1,reinas)  

18     }  

19  

20     def coloqueReinasHastaLaFila(k:Int):Set[List[Int]] = {  

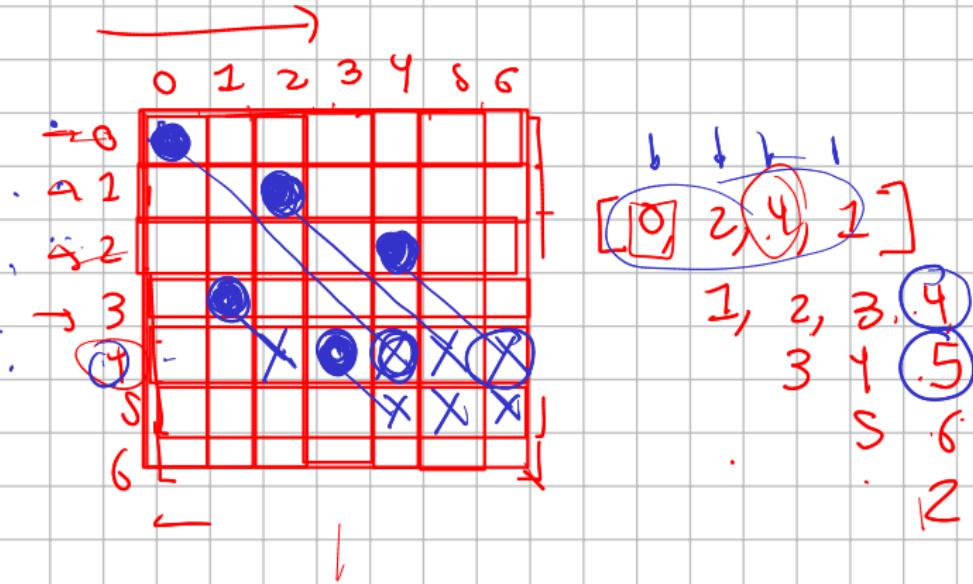
21       ...  

22     }  

23     coloqueReinasHastaLaFila(n)  

24   }

```



# Contents

## 1 Secuencias

- Vectores, cadenas y rangos
- Expresiones For y Búsqueda Combinatoria

## 2 Conjuntos

- Expresiones For
- Conjuntos

## 3 Expresiones For

- Consultas con For
- Traducción del For
- Generalización del For

## 4 Maps o Asociaciones

## 5 Streams o Flujos

- Flujos
- Evaluación perezosa
- Calculando con secuencias infinitas

# Consultas con *For*

- Las expresiones *For* son especialmente útiles para realizar consultas típicas de bases de datos.
- **Ejemplo:** Suponga que tenemos una base de datos de libros, representada por una lista de libros de Scala:

```
0 case class Libro(titulo:String, autores>List[String])
1 val libros:List[Libro] = List(
2   Libro(titulo = "Structure-and-Interpretation-of-Computer-Programs",
3     autores= List("Abelson,-Harald", "Sussman,-Gerald-J.")),
4   Libro(titulo= "Programming-in-Scala",
5     autores=List("Odersky,-Martin", "Spoon,-Lex", "Venners,-Bill")),
6   Libro(titulo="Learning-Concurrent-Programming-in-Scala",
7     autores=List("Prokopec,-Aleksandar")),
8   Libro(titulo= "Working-Hard-to-Keep-it-Simple",
9     autores=List("Odersky,-Martin")),
10 )
```

- Algunas consultas típicas son:
  - Listar los libros cuyo autor es *Odersky*
  - Listar los libros con la palabra "*Program*" en el título
  - Liste los autores en la base de datos que han escrito al menos dos libros

## Consultas con *For* (2)

- Listar los libros cuyo autor es *Odersky*

```
0 // Libros cuyo autor sea Odersky
1 for (l<-libros; a<-l.autores if a startsWith("Odersky , "))
2     yield l.titulo
```

- Listar los libros con la palabra "*Program*" en el título

```
0 // Libros con la palabra Program en el titulo
1 for (l<-libros if (l.titulo indexOf "Program")>=0)
2     yield l.titulo
```

- Liste los autores en la base de datos que han escrito al menos dos libros

```
0 // Autores que hayan escrito al menos 2 libros en la base de datos (aunque salgan repetidos)
1 for {
2     l1<-libros
3     l2<-libros
4     if l1!=l2
5     a1<-l1.autores
6     a2<-l2.autores
7     if a1==a2
8 } yield a1
```

# Consultas con *For* (3)

- Liste los autores en la base de datos que han escrito al menos dos libros, pero que no salgan repetidos

```
0 // Autores que hayan escrito al menos 2 libros en la base de datos
1 // eliminando repetidos
2 (for {
3   l1<-libros
4   l2<-libros
5   if l1!=l2
6   a1<-l1.autores
7   a2<-l2.autores
8   if a1==a2
9 } yield a1).distinct
```

- Una solución usando conjuntos:

```
0 // Autores que hayan escrito al menos 2 libros en la base de datos
1 // eliminando repetidos por usar conjuntos y no listas
2 val conjuntoLibros = libros.toSet
3 for {
4   l1<-conjuntoLibros
5   l2<-conjuntoLibros
6   if l1!=l2
7   a1<-l1.autores
8   a2<-l2.autores
9   if a1==a2
10 } yield a1
```

# Expresiones *For* y funciones de alto orden

- Ya vimos que las expresiones *For* son una abstracción lingüística que esconde el uso de *map*, *flatmap* y *filter*.
- De hecho, si las expresiones *For* fueran nucleares, podríamos escribir *map*, *flatmap* y *filter* así:

```
0 def mapFun[T, U] (xs: List[T], f: T => U): List[U] =  
1   for (x <- xs) yield f(x)  
2  
3 def flatMap[T, U] (xs: List[T], f: T => Iterable[U]): List[U] =  
4   for (x <- xs; y <- f(x)) yield y  
5  
6 def filter[T] (xs: List[T], p: T => Boolean): List[T] =  
7   for (x <- xs if p(x)) yield x
```

# Traducción de expresiones For (1)

- En la realidad, el compilador de Scala traduce las expresiones *For* en términos de *map*, *flatMap* y una variante perezosa de *filter*.
- Por ejemplo, para el caso sencillo siguiente

```
0  for (x <-e1) yield e2
```

la traducción es:

```
0  e1.map(x => e2)
```

- Para el caso siguiente:

```
0  for (x <-e1 if f; s) yield e2
```

donde *f* es un filtro y *s* es una secuencia (puede ser vacía) de generadores y filtros, la traducción es:

```
0  for (x <-e1.withFilter(x => f); s) yield e2
```

y la traducción continúa con la nueva expresión.

# Traducción de expresiones For (2)

- Para el caso siguiente:

```
0   for (x <- e1; y <- e2; s) yield e3
```

donde *s* es una secuencia (puede ser vacía) de generadores y filtros, la traducción es:

```
0   e1.flatMap(x => for (y <- e2; s) yield e3)
```

y la traducción continúa con la nueva expresión.

# Traducción de expresiones For (3)

- Recordemos nuestro ejemplo previo: Dado un entero positivo  $n$ , genere todas las parejas de enteros positivos  $(i,j)$  tal que  $1 \leq j < i < n$  y  $i + j$  es primo.

```
0 for {
1   i <- 2 until n
2   j <- 1 until i
3   if esPrimo(i+j)
4 } yield (i,j)
```

Este se traduce en:

```
0 (2 until n).flatMap (i =>
1   (1 until i).withFilter(j => esPrimo(i+j))
2     .map (j => (i,j)))
```

# Generalización del *For*

- Nótese que la traducción del *For* no se limita a listas, secuencias o colecciones.
- La traducción está basada en contar con los métodos *map*, *flatMap* y *withFilter*.
- Esto permite usar la sintaxis de expresiones *For* para sus propios tipos siempre y cuando les defina los métodos *map*, *flatMap* y *withFilter*.
- Hay diversidad de tipos donde esto es muy útil: arreglos, iteradores, bases de datos, datos XML, analizadores, ...

# Contents

## 1 Secuencias

- Vectores, cadenas y rangos
- Expresiones *For* y Búsqueda Combinatoria

## 2 Conjuntos

- Expresiones *For*
- Conjuntos

## 3 Expresiones *For*

- Consultas con *For*
- Traducción del *For*
- Generalización del *For*

## 4 Maps o Asociaciones

## 5 Streams o Flujos

- Flujos
- Evaluación perezosa
- Calculando con secuencias infinitas

# Maps o Asociaciones

- Otro tipo de colección fundamental son las asociaciones o *maps*.
- Una asociación de tipo *Map[LLave, Valor]* es una estructura de datos que asocia claves de tipo *LLave* con valores de tipo *Valor*.

```
0  val numerosRomanos = Map("I" -> 1, "V" -> 5, "X" -> 10)
1  val capitalDePais = Map("USA" -> "Washington", "COLOMBIA" -> "Bogota")
```

- La clase *Map[LLave, Valor]* extiende la clase *Iterable[(LLave, Valor)]*

```
0  scala> val paisDeCapital = capitalDePais map { case (x,y) => (y,x) }
1  val paisDeCapital: scala.collection.immutable.Map[String, String] =
2    Map(Washington -> USA, Bogota -> COLOMBIA)
```

Nótese que las asociaciones son extensión de la clase Iterable parametrizada en parejas de tipo (*LLave, Valor*).

La sintaxis *llave -> valor* es azúcar sintáctico de la sintaxis (*llave, valor*)

## Los maps son funciones

- Las asociaciones son también extensión del tipo  $LLave \Rightarrow Valor$ , por tanto pueden ser utilizados donde las funciones puedan ser usadas.
- En particular, los maps se pueden aplicar a argumentos de tipo  $LLave$ :

```
0  capitalDePais ("COLOMBIA" )
```

- Si no hay nada asociado con esa llave, se genera un error

```
0  scala> capitalDePais("US")
1  java.util.NoSuchElementException: key not found: US ...
```

- Hay una operación *withDefaultValue* que convierte una asociación en una función total:

```
0  scala> val cap1 = capitalDePais withDefaultValue "Desconocida"
1  val cap1: scala.collection.immutable.Map[String ,String ] = Map(USA \>> Washington , COLOMBIA \>> Bogota
2
3  scala> cap1("US")
4  val res1: String = Desconocida
```

# Contents

## 1 Secuencias

- Vectores, cadenas y rangos
- Expresiones *For* y Búsqueda Combinatoria

## 2 Conjuntos

- Expresiones *For*
- Conjuntos

## 3 Expresiones *For*

- Consultas con *For*
- Traducción del *For*
- Generalización del *For*

## 4 Maps o Asociaciones

## 5 *Streams o Flujos*

- Flujos
- Evaluación perezosa
- Calculando con secuencias infinitas

# Colecciones y búsqueda combinatoria

- Hemos visto un buen número de colecciones inmutables que proveen poderosas operaciones, en especial para operaciones combinatorias.
- Por ejemplo, para encontrar el segundo número primo entre 1000 y 10000:

```
0 def esPrimo(n:Int):Boolean = (2 until n).forall(n% != 0)
1 ((1000 to 10000) filter esPrimo)(1)
```

- Esto se escribe mucho más corto que la alternativa recursiva:

```
0 def enesimoPrimo(desde:Int, hasta:Int, n:Int): Int=
1   if (desde >= hasta) throw new Error("No-hay-al-menos-dos-primos")
2   else if (esPrimo(desde))
3     if (n==1) desde
4     else enesimoPrimo(desde+1, hasta, n-1)
5   else   enesimoPrimo(desde+1, hasta, n)
6
7 def segundoPrimo(desde:Int, hasta: Int)= enesimoPrimo(desde, hasta, 2)
8 segundoPrimo(1000,10000)
```

# Problemas de desempeño

- Desde un punto de vista de desempeño, la solución:

$$((1000 \text{ to } 10000) \text{ filter } \text{esPrimo})(1)$$

es bastante mala; calcula **todos los primos** entre 1000 y 10000, para luego sólo necesitar el segundo elemento de esa lista.

- Podríamos tratar de disminuir la cota superior pero no sabemos hasta donde podemos hacerlo (puede que perdamos el segundo primo de la lista si disminuimos mucho)
- Otra idea, mucho mejor, es lograr que cuando se construyan listas, no se calculen explícitamente las colas de las listas, mientras no las necesite algún otro cálculo.
- Esta idea se implementa en la clase *Stream* que crea una colección denominada **Flujos**.

Los flujos son como las listas, pero su cola es evaluada **por demanda**

# Definiendo flujos

- Los flujos se definen a partir de un flujo vacío `Stream.empty` y de un constructor de flujos `Stream.cons`.

```
0 val xs=Stream.cons(1, Stream.cons(2, Stream.empty))
```

- En nuevas versiones se usa `LazyList` en lugar de `Stream`
- Podemos crear rangos perezosos y calcular con ellos:

```
0 def streamRange(min:Int, max:Int):LazyList[Int] =  
1   if (min>=max) LazyList.empty  
2   else LazyList.cons(min, streamRange(min+1, max))  
3  
4 val primos = (streamRange(1000,10000) filter esPrimo)  
5 primos(2)  
6 primos  
7 primos(4)  
8 primos
```

Nótese que **cada que se necesita** un valor nuevo, se calculan nuevos elementos del rango

# Métodos de flujos

- *LazyList* soporta casi todos los operadores de las listas.
- La principal excepción es `::`. Este operador produce siempre listas, nunca flujos.
- Existe un operador alternativo: `# ::`

```
0 1 #:: 2 #:: LazyList.empty
```

- `# ::` se puede usar tanto en expresiones como en patrones.

# Evaluación perezosa

- Scala usa evaluación ansiosa (o estricta) por defecto, pero permite la evaluación perezosa, usando *lazy val*:

*lazy val* = *expr*

- Por ejemplo:

```
0 def expr = {  
1   val x:Int ={print("x"); 1}  
2   lazy val y:Int = {print("y"); 2}  
3   val z:Int = {print("z"); 3}  
4   z + y + x + z + y +x  
5 }  
6 expr
```

¿Al ejecutar este programa qué se imprime mientras se calcula *expr*?

# Flujos infinitos

- La evaluación perezosa permite modelar estructuras de datos infinitas.
- Por ejemplo, podemos calcular el conjunto de todos los enteros:

```
0  scala> def enterosDesde(n:Int) : LazyList[Int] = n #:: enterosDesde(n+1)
1  def enterosDesde(n: Int): LazyList[Int]
2
3  scala> val enteros = enterosDesde(0)
4  val enteros: LazyList[Int] = LazyList(<not computed>)
5
6  scala> enteros (5)
7  val res1: Int = 5
8
9  scala> enteros
10 val res2: LazyList[Int] = LazyList(0, 1, 2, 3, 4, 5, <not computed>)
```

# La criba de Eratóstenes

La criba de Eratóstenes es un método antiguo para calcular los números primos:

- Comience con la lista de todos los enteros a partir de 2, el primer número primo
- Tome el ‘primer elemento de la lista (es primo). Y filtre el resto de la lista, eliminando los que sean múltiplos de ese primer primo.
- El primer elemento de la nueva lista es un primo. Filtre el resto de esa lista, nuevamente eliminando los que sean múltiplos de ese primer primo.
- El primer elemento de la nueva lista es un primo. Y así sucesivamente. Itere por siempre.

Este proceso es imposible de implementarlo con evaluación ansiosa

# La criba de Eratóstenes en Scala

```
0 scala> def criba(s:LazyList[Int]): LazyList[Int] =  
1           s.head #:: criba(s.tail filter (_ % s.head != 0))  
2 def criba(s: LazyList[Int]): LazyList[Int]  
3  
4 scala> val primos = criba(enterosDesde(2))  
5 val primos: LazyList[Int] = LazyList(<not computed>)  
6  
7 scala> primos(20)  
8 val res3: Int = 73  
9  
10 scala> primos  
11 val res4: LazyList[Int] = LazyList(2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53,  
12                                         59, 61, 67, 71, 73, <not computed>)
```

# Conclusiones sobre la programación funcional

La programación funcional provee un conjunto coherente de conceptos y técnicas de programación basados en:

- Programación de alto orden
- Reconocimiento de patrones
- Colecciones inmutables
- Ausencia de estado explícito
- Estrategias de evaluación flexible: ansiosa/perezosa/por nombre

La programación funcional provee una caja de herramientas útiles para todo programador.

La programación funcional provee una forma diferente de pensar para programar