Tema 6: Programación Funcional en Scala

En este tema vamos a repasar algunas características de Programación Funcional de Scala, que hacen posible utilizar este paradigma en un lenguaje moderno, orientado a objetos y que corre en la máquina virtual Java.

Contenidos

1. Introducción

1.1. Programación funcional en Scala

Como ya hemos visto en temas previos, en programación funcional se definen funciones al estilo matemático, funciones que transforman un valor en otro sin mantener ningún estado interno. Dado un conjunto de valores de entrada, la función debe siempre devolver la misma salida.

En programación funcional, además, no existen variables mutables que puedan modificar su valor de un paso de ejecución a otro. No existe ni siquiera el concepto de paso de ejecución. Todas las computaciones se realizan tomando objetos inmutables y transformándolos mediante funciones en otros nuevos objetos construidos.

En palabras de *Scala in Action*:

> Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data.

Scala integra estas y otras características funcionales en un lenguaje estrictamente tipeado de Programación Orientada a Objetos construyendo un completo lenguaje que permite al desarrollador utilizar las características más adecuadas a cada problema.

En Scala es posible definir funciones que no estén ligadas a clases ni objetos. Estas funciones pueden utilizarse en cualquier método de cualquier clase o en otras funciones.

Gracias a las funcionalidades de Scala es posible definir programas que aprovechan las posibilidades de las arquitecturas paralelas (como CPUs multi-nucleos) y que son fácilmente escalables a sistemas concurrentes con múltiples servidores, procesadores e hilos de ejecución.

1.2. Características funcionales de Scala

Algunas técnicas de programación funcional que hemos visto en Scheme y que veremos en Scala

- definición de funciones de forma declarativa, sin utilizar pasos de ejecución
- valores inmutables, no existen efectos laterales en los programas
- uso de la recursión para definir funciones que deben hacer iteraciones
- uso de listas
- funciones como objetos de primera clase
- funciones de orden superior
- clousures: funciones con variables libres que se instancian en tiempo de ejecución

En Scala existen bastantes características de programación funcional que no hemos visto en Scheme. Veremos algunas de ellas en este tema:

- funciones y las variables tipeadas
- definición por comprensión de colecciones
- currying

2. Características básicas de Scala

2.1. Variables y funciones *tipeadas*

Como hemos visto en el seminario, Scala es un lenguaje strictamente tipeado. Todas las variables, funciones y parámetros tienen un tipo definido, y el compilador detecta un error cuando usamos un valor de un tipo que no corresponde al tipo de la variable o del parámetro. También vimos en el seminario que no es necesario especificar el tipo devuelto por la función porque el compilador de Scala lo puede inferir a partir del tipo devuelto por la sentencia que implementa la función. Sin embargo, por claridad, vamos a incluirlo en las definiciones que vamos a ver en este apartado.

2.1.1. Funciones

Por ejemplo, la función `max`

```
def max(x: Int, y: Int): Int = if (x > y) x else y
```

recibe dos objetos `Int` y devuelve un `Int`. Esta firma se denota en Scala de la siguiente forma:

```
(Int,Int) => Int
```

Se declaran entre paréntesis los tipos de los argumentos y después de la flecha el tipo del resultado de la función. Más adelante veremos que Scala utiliza esta notación para definir el tipo de las funciones anónimas.

2.1.2. Tuplas

Los tipos de las tuplas se definen con un paréntesis y los tipos de los elementos entre comas:

```
val t: (Int, String, Double) = (21, "Veintiuno", 21.0)
```

2.1.3. Colecciones

Las colecciones se definen en Scala como clases genéricas en las que es necesario definir el tipo de los elementos contenidos. Para ello se debe definir entre corchetes el tipo de los valores contenidos en el dato compuesto.

Ejemplos:

```
val miListaInt: List[Int] = List(1,2,3)
val miListaString: List[String] = List("Hola","que","tal")
```

2.1.4. ¿Qué es *Any*?

El tipo `Any` es el tipo (o clase) padre de todos los tipos de Scala, es similar a la clase `Object` de Java. Cualquier objeto es de ese tipo.

Por ejemplo, las siguientes sentencias son correctas en Scala:

```
val s: String = "Hola"
val cualquierTipo: Any = s
```

La variable `cualquierTipo` es de tipo `Any` y puede contener cualquier tipo de valor. La clase `Any` es padre de cualquier otra clase de Scala.

En el siguiente ejemplo, declaramos una lista que contiene distintos tipos de valores:

```
val listaCualquiera: List[Any] = List("Hola",1,"Adios",2)
```

Los métodos `isInstanceOf` y `asInstanceOf` comprueban si el valor almacenado en una variable de tipo `Any` es de otro tipo y permiten recuperarlo como un objeto de ese tipo:

```
val cualquierTipo: Any = "Hola"
cualquierTipo.isInstanceOf[String]
⇒ Boolean = true
cualquierTipo.asInstanceOf[String]
⇒ String = Hola
```

También es posible hacer comprobar el tipo original de un objeto usando la sentencia `match`:

```
val p: Any = "Hola"
p match {
        case s: String => s+s
        case s: Int => "Int"
        case _ => "Otro tipo"
}
⇒ String = "HolaHola"
```

El `match` anterior devuelve la cadena `"HolaHola"`.

2.2. Definición de funciones de forma declarativa

Cuando programamos en Scala utilizando el paradigma funcional no debemos utilizar bucles ni pasos de ejecución. Al igual que hacíamos en Scheme, la función debe quedar definida en una única sentencia. Eso sí, en esa sentencia pueden haber anidadas llamadas a otras funciones o sentencias adicionales. Por ejemplo, una función que devuelve el máximo de 3 números se puede definir de la siguiente forma, llamando a la función `max` definida previamente:

```
def max3(x: Int, y: Int, z: Int): Int = max(max(x,y),z)
```

Permitimos definir variables locales inmutables al comienzo de las definiciones de funciones:

```
def max3(x: Int, y: Int, z: Int): Int = {
   val maxPrimeros = max(x,y)
   max(maxPrimeros,z)
}
```

2.3. Valores inmutables

En Scala se recomienda utilizar variables de tipo `val` para reforzar el carácter funcional y evitar los efectos laterales en los programas. Una variable de tipo `val` no puede ser modificada (re-asignada a otro objeto).

Por ejemplo, las siguientes sentencias darían un error en un script de Scala:

```
val numVal = 1
numVal = numVal + 1 //error!!
```

Como ya hemos visto en el seminario, Scala también permite declarar variables que no tienen esta restricción utilizando la palabra clave `var` :

```
var numVar = 1
numVar = numVar + 1 // no hay error
```

Independientemente del tipo de mutabilidad de las variables, en Scala también se diferencia entre tipos mutables y tipos inumutables. Las estructuras de datos inmutables no pueden ser modificadas una vez creadas. Los tipos mutables si pueden modificarse.

Por ejemplo, la clase `List` es inmutable, mientras que la clase `Array` es mutable. En el siguiente ejemplo vemos cómo modificar el valor de una posición de un array.

```
val numbers = Array(1, 2, 3, 4)
val num = numbers(3)
numbers(3) = 100
val numCambiado = numbers(3)
```

La declaración de una variable como `var` o `val` no tiene nada que ver con la mutabilidad del objeto que hay alojado en ella. En el caso anterior, la variable `numbers` es de tipo `val` y puede almacenar un array que es modificado.

2.4. Listas

En el seminario hemos visto que Scala tiene operaciones sobre listas similares a las de Scheme. La clase `List` es una estructura inmutable a la que podemos aplicar métodos como:

- `Nil` : lista vacía
- `head` : devuelve el primer elemento de la lista
- `tail` : devuelve el resto
- `isEmpty` : comprueba si la lista es vacía
- `::` : devuelve una nueva lista añadiendo un elemento a su cabeza (similar a `cons`)
- `:::` : devuelve una nueva lista concatenando dos listas (similar a `append`)

En la clase `List` también se definen un conjunto de métodos de orden superior como `exists` , `filter` , `fold` o `map` que veremos más adelante.

Ejemplos:

```
val lista: List[Int] = List(1, 2, 3, 4)
val primero = lista.head
val resto = lista.tail
val dobleLista = primero :: resto ::: lista
```

2.5. Uso de la recursión

Ya hemos visto en el seminario algunos ejemplos de funciones recursivas. El uso de la recursión en Scala es similar al que hemos visto en Scheme. La única diferencia es que en Scala debemos definir los tipos de las funciones y de sus parámetros.

Ejemplos de recursión:

- Lista palíndroma
- Fibonacci recursivo
- Factorial iterativo como ejemplo de recursión por la cola
- Fibonacci iterativo como ejemplo de recursión por la cola

Número de elementos de una lista:

```
def numElems(lista:List[Int]):Int =
   if (lista.isEmpty) 0 else
```

```
      1 + numElems(lista.tail)
```

Por ejemplo:

```
numElems(List(1,2,3,4,5))
⇒ Int = 5
```

Inserción en una lista ordenada:

```
def insert(x: Int, lista: List[Int]) : List[Int] =
   if (lista == Nil) x :: Nil else
      if (x < lista.head) x :: lista else
         lista.head :: insert(x, lista.tail)
```

Por ejemplo:

```
insert(10, List(-10, -2, 0, 30, 54))
⇒ List[Int] = List(-10, -2, 0, 10, 30, 54)
```

Ordenación de una lista:

```
def sort(lista: List[Int]): List[Int] =
   if (lista.isEmpty) Nil else
      insert(lista.head, sort(lista.tail))
```

Por ejemplo:

```
sort(List(20,-1,3,34,-10))
⇒ List[Int] = List(-10, -1, 3, 20, 34)
```

Y, por último, inversión de una lista:

```
def reverse(lista: List[Int]) : List[Int] =
   if (lista.isEmpty) Nil else
      reverse(lista.tail) ::: List(lista.head)
```

Por ejemplo:

```
reverse(List(2,5,10,12))
⇒ List[Int] = List(12, 10, 5, 2)
```

Para familiarizarse con el lenguaje recomendamos rehacer en Scala los ejemplos y ejercicios

vistos en Scheme.

Todas las funciones anteriores son sobre listas de enteros. Scala es un lenguaje fuertemente tipeado, lo que obliga a definir los tipos de los elementos que hay en las listas.

Es posible definir funciones más genéricas utilizando tipos genéricos. Veremos más adelante un ejemplo, y en más profundidad cuando veamos Programación Orientada a Objetos en Scala.

3. Funciones como objetos de primera clase

Scala cumple con una de las características fundamentales de los lenguajes funcionales como es el uso de funciones como objetos de primera clase. En Scala es posible definir una función en tiempo de ejecución y asignarla a una variable, pasarla como parámetro a otra función o devolverla como el resultado de invocar otra función.

3.1. El tipo función

Cada función tiene un *tipo función* específico, formado por los tipos de los parámetros y el tipo devuelto como función.

Por ejemplo:

```
def sumaDosInts(x: Int, y: Int): Int = {
    x + y
}
def multiplicaDosInts(x: Int, y: Int): Int = {
    x*y
}
```

Estamos definiendo dos funciones sencillas llamadas `sumaDosInts` y `multiplicaDosInts`. Estas funciones toman dos valores `Int` y devuelven un valor `Int`, resultante de ejecutar la operación matemática apropiada.

El tipo de ambas funciones es `(Int, Int) => Int`.

Como en cualquier lenguaje funcional, en Scala puedes usar los tipos función como cualquier otro tipo. Por ejemplo, es posible definir una variable del tipo función y asignarle una función apropiada:

```
var funcionMatematica: (Int, Int) => Int = sumaDosInts
```

Esto puede leerse de la siguiente forma:

> Define una variable llamada `funcionMatematica`, que tiene el tipo de una función que toma dos valores `Int` y devuelve un valor `Int`. Establece que esta variable se

> refiera a la función llamada `suma`.

La función `sumaDosInts` tiene el mismo tipo que la variable `funcionMatematica` por lo que es posible realizar la asignación sin que el comprobador de tipos de Scala de un error.

Ahora podemos llamar a la función asignada usando el nombre `funcionMatematica`:

```
funcionMatematica(3,10) ⇒ 13
```

Podemos asignar una función diferente con el mismo tipo a la misma variable, de la misma forma que hacemos con tipos no función:

```
funcionMatematica = multiplicaDosInts
funcionMatematica(3,10) ⇒ 30
```

cualquier otro tipo, es posible hacer que Scala infiera el tipo de la variable, pero hay que indicarle al compilador de Scala con un subrayado (`_`) que queremos usar la función tal cual, sin invocarla. Si no utilizamos el subrayado el compilador se quejará diciendo que faltan los parámetros de `sumaDosInts`.

```
val otraFuncionMatematica = sumaDosInts _
```

El uso del subrayado (`_`) es muy común es Scala como *hueco* (*place holder*) que indica *cualquier cosa*. Por ejemplo, lo podemos usar para construir funciones nuevas a partir de funciones ya existentes. Supongamos la siguiente función:

```
def sumaTresInt(x: Int, y: Int, z: Int): Int = {
    x + y + z
}
```

Podemos asignar esta función a una variable de tipo `(Int) => Int` fijando dos de sus parámetros y dejando libre con un `_` el tercero:

```
val suma10y8: (Int) => Int = sumaTresInt(10, 8 , _: Int)
```

Se puede leer de la siguiente forma:

> Define una constante `suma10y8` del tipo función que toma un `Int` y devuelve un `Int` en la que se referencia a una función resultante de fijar el primer y el segundo parámetro a los valores 10 y 8 y dejar el último parámetro como el único de la función.

```
suma10y8(3) ⇒ 21
```

3.2. Funciones como parámetros

Podemos pasar una función como parámetro de otra. Veamos el ejemplo del sumatorio de una función que ya resolvimos en Scheme:

```
sumatorio(f, a, b) = f(a) + sumatorio(f, a + 1, b)
sumatorio(f, a, b) = 0 si a > b
```

En Scala:

```
def sumatorio(a: Int, b:Int, f:(Int) => Int): Int = {
   if (a > b) 0
   else f(a) + sumatorio(a + 1, b, f)
}
```

La función `sumatorio` tienen como tercer parámetro una función con la signatura `(Int)=>Int`, que admite un entero como parámetro y devuelve entero. Podemos pasar como parámetro cualquier función definida de este tipo. Por ejemplo, cualquiera de las siguientes funciones:

```
def cuadrado(x: Int) = x * x
def cubo(x: Int) = x * x * x
def suma3(x: Int) = x + 3
```

Las llamadas serían de la siguiente forma:

```
sumatorio(1, 10, cuadrado) ⇒ 385
sumatorio(1, 10, cubo) ⇒ 3025
sumatorio(1, 10, suma3) ⇒ 85
```

3.3. Funciones anónimas

En Scala es posible crear funciones anónimas. Al igual que en Scheme, las expresiones que construyen una función anónima se denominan *expresiones lambda* (aunque la palabra “lambda” no aparezca en ninguna parte de la expresión). La sintaxis es la siguiente:

```
(parámetros) => cuerpo
```

Por ejemplo:

```
(x: Int, y: Int) => {x + y}
```

Con esta expresión definimos un *objeto función* que tiene dos argumentos de tipo `Int`. Después de la flecha definimos el cuerpo de la función, en este caso la suma de los dos argumentos.

Si ejectuamos en Scala la sentencia vemos que devuelve como resultado un valor de tipo `(Int, Int) => Int`.

```
(x: Int, y: Int) => {x + y}
⇒ (Int, Int) => Int = <function2>
```

Este debe ser el tipo de la variable o parámetro al que queramos asignar el objeto función:

```
val fun: (Int, Int) => Int = (x: Int, y: Int) => {x + y}
```

Una vez asignada a una variable, podemos aplicar la función de la forma habitual:

```
fun(2,3) ⇒ Int = 5
```

Veamos otro ejemplo. Podemos utilizar una función anónima como tercer parámetro de la función `sumatorio`. Para calcular el sumatorio desde 1 hasta 10 de la función `x+10`:

```
sumatorio(1, 10, (x:Int) => {x+10}) ⇒ Int = 155
```

3.3.1. Abreviaturas de las expresiones lambda

Podemos utilizar las características de inferencia de tipos del compilador y no especificar el tipo del parámetro de la función anónima. El compilador sabe que es un `Int` por el tipo del tercer argumento de `sumatorio` (la signatura `(Int) => Int`):

```
sumatorio(1, 10, (x) => {x+10})
```

Podemos ser más concisos todavía quitando los paréntesis y las llaves:

```
sumatorio(1, 10, x => x+10)
```

Y más todavía con la sintaxis de huecos:

```
sumatorio(1, 10, _+10)
```

Las dos últimas expresiones son equivalentes. El compilador transforma la última en la anterior.

3.4. Funciones en listas

Al igual que en Scheme es posible guardar funciones en otras estructuras. Por ejemplo, podemos crear una lista de funciones, pero todas las funciones deben tener la misma signatura, que será el tipo de los elementos de la lista:

```
def suma3(a: Int, b: Int, c: Int) = a + b + c
def mult3(a: Int, b: Int, c: Int) = a * b * c
val listaFuncs: List[(Int,Int,Int)=>Int] = List(suma3 _, mult3 _, (x:Int,y:Int,z:Ir
val f = listaFuncs.head
f(1,2,3)
⇒ Int = 6
```

Ejemplo de función que toma una lista de funciones de un argumento y las aplica a un número de forma correlativa:

```
def aplicaLista (lista: List[(Int) => Int], x: Int): Int = {
   if (lista.length == 1) lista.head(x)
   else lista.head(aplicaLista(lista.tail,x))
}
def mas5(x: Int) = x+5
def por8(x: Int) = x*8

val l = List(mas5 _, por8 _, suma3(1, _: Int, 10))
aplicaLista(l, 10)
⇒ Int = 173
```

En el ejemplo creamos una función de un argumento a partir de la función de 3 argumentos `suma3` utilizando la sintaxis de huecos.

3.5. Funciones como valores devueltos

Las funciones pueden crearse en la invocación de otras funciones y ser devueltas como resultado. Nos estamos acercando al concepto de *clausura* que vimos en Scheme y que veremos en un par de apartados.

Por ejemplo, la siguiente función construye y devuelve un sumador que suma el número que pasamos como parámetro. La función devuelta es una clausura.

```
def makeSumador(k: Int) = {
   (x: Int) => x + k
}
val f: (Int) => Int = makeSumador(10)
val g: (Int) => Int = makeSumador(100)
f(4)
⇒ Int = 14
```

```
g(4)
⇒ Int = 104
```

La definición de `makeSumador` en Scheme sería:

```
(define (make-sumador k)
   (lambda (x) (+ x k)))
```

La función `makeSumador` devuelve una función, por lo que podríamos haber declarado su tipo como

```
def makeSumador(k: Int): (Int) => Int = { ... }
```

Pero hemos dejado que sea el compilador el que lo infiera. Sin embargo, en las variables `f` y `g` lo hemos indicado por claridad.

Podemos invocar a la función devuelta por `makeSumador` directamente sin utilizar las variables `f` o `g` utilizando la siguiente expresión:

```
makeSumador(10)(100)
⇒ Int = 110
```

Sería equivalente a la siguiente expresión en Scheme:

```
((makesumador 10) 100)
```

4. Funciones de orden superior

La definición de funciones como un tipo de dato primitivo permite definir *funciones de orden superior* que toman como parámetro otras funciones y permiten hacer mucho más conciso y general el código escrito en Scala.

4.1. FOS en la clase `List`

Por ejemplo, en la clase `List` se definen métodos como los siguientes, que admiten funciones como parámetro.

- `count` : cuenta el número de elementos de la lista que satisfacen un predicado
- `exists` : comprueba si algún elemento de la lista satisface un predicado
- `filter` : devuelve una nueva lista con los elementos de la lista original que cumplen el predicado
- `map` : aplica una función a todos los elementos de la lista, devolviendo una nueva lista con el resultado

- `foldRight` : pliega la lista con una función de plegado, devolviendo un único dato como resultado

Son métodos definidos como genéricos que se pueden aplicar a listas de distintos tipos. El tipo de la función que se pasa como argumento tiene que corresponder con el tipo de los elementos de la lista.

Veamos distintas formas de llamar a estas funciones genéricas, utilizando `count` como ejemplo:

```
val lista = List(1,2,3,4,5,6,7,8,9)
lista.count((x:Int) => {x % 2 == 0})
⇒ Int = 4
```

Podemos escribir la función que pasamos como parámetro de `count` de distintas formas, cada una de ellas más concisa. El compilador de Scala se encarga de completar lo necesario:

```
lista.count((x) => {x % 2 == 0})
lista.count(x => x % 2 == 0)
lista.count(_ % 2 == 0)
```

podemos definir la función `par` y pasarla como parámetro:

```
def par(x:Int): Boolean = {x % 2 == 0}
lista.count(par)
```

Algunos ejemplos más:

```
lista.filter(par) ⇒ List[Int] = List(2, 4, 6, 8)
def cuadrado(x:Int) = x*x
lista.map(cuadrado) ⇒ List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81)
List("No","es","elegante","escribir","con","mayusculas").map(s => s.toUpperCase)
⇒ List[String] = List(NO, ES, ELEGANTE, ESCRIBIR, CON, MAYUSCULAS)
```

Al igual que hacíamos en Scheme, es importante no sólo saber utilizar estas funciones sino también su implementación recursiva.

Función `miCount` , que cuenta el número de elementos de una lista de tipo `Int` que cumplen un predicado:

```
def miCount(lista:List[Int], p:(Int)=>Boolean): Int = {
        if (lista == Nil) 0
    else if (p(lista.head)) 1+miCount(lista.tail,p)
    else miCount(lista.tail,p)
}
```

Función `miMap`, que aplica una función a todos los elementos de una lista:

```
def miMap(lista:List[Int], f:(Int)=>Int): List[Int] = {
        if (lista == Nil) Nil
    else f(lista.head) :: miMap(lista.tail,f)
}
```

4.2. Funciones genéricas

En Scala también es posible definir funciones con tipos de datos genéricos. Por ejemplo, la siguiente función `miFilter` utiliza el tipo genérico `A` como el tipo de los elementos de la lista y el tipo del predicado que se aplica a los elementos.

El compilador determina el tipo `A` cuando se invoca a la función con una lista y un predicado concreto.

```
def miFilter[A](lista: List[A], pred: (A)=>Boolean): List[A] = {
        if (lista.isEmpty) Nil
    else if (pred (lista.head)) lista.head :: miFilter(lista.tail, pred)
        else miFilter(lista.tail, pred)
}

def par(x: Int) = x % 2 == 0

miFilter(List(1,2,3,4), par)
⇒ List[Int] = List(2, 4)
miFilter(List("hola","amigo","adios"),(s: String) => { s.head == 'a'})
⇒ List[String] = List(amigo, adios)
```

Las versiones genéricas de las funciones recursivas sobre listas que vimos anteriormente son:

```
def numElems[A](lista:List[A]):Int = {
   if (lista.isEmpty) 0
   else 1 + numElems(lista.tail)
}

def reverse[A](lista: List[A]) : List[A] = {
   if (lista.isEmpty) Nil
   else reverse(lista.tail) ::: List(lista.head)
}

def insert[A](x: A, lista: List[A], menor: (A,A) => Boolean) : List[A] = {
   if (lista == Nil) x :: Nil
   else if (menor(x,lista.head)) x :: lista
   else lista.head :: insert(x, lista.tail, menor)
}
```

```
def sort[A](lista: List[A], menor: (A,A) => Boolean): List[A] = {
   if (lista.isEmpty) Nil
   else insert(lista.head, sort(lista.tail, menor), menor)
}
```

Por ejemplo, podemos definir una ordenación entre tuplas de dos enteros basada en comparar la suma de sus dos componentes. Y después, utilizarla para ordenar una lista de tuplas:

```
def menorTuplas(t1: (Int,Int), t2: (Int,Int)) = t1._1+t1._2 < t2._1+t2._2
sort(List((1,2),(3,1),(-1,-2)), menorTuplas)
⇒ List[(Int, Int)] = List((-1,-2), (1,2), (3,1))
```

4.3 Métodos `foldRight` y `foldLeft`

Hay que hacer una mención especial al los métodos de orden superior `foldRight` y `foldLeft` de la clase `List`. Son similares a los que vimos en Scheme: método de plegado, que recorren una lista de derecha a izquierda, y de izquierda a derecha, aplicando una función que transforma los elementos de dos elementos en uno. Los dos elementos son el resultado anterior (o el caso base, cuando la lista es vacía), y el elemento de la lista.

Sus definiciones son las siguientes (siendo *A* el tipo de los elementos de la lista, y *B* el tipo del caso baso):

```
def foldRight[B](z: B)(f: (A, B) => B): B
def foldLeft[B](z: B)(f: (B, A) ⇒ B): B
```

En la definición se utiliza un *currying* (lo veremos más adelante). No nos preocupemos ahora mismo los dos paréntesis, tenemos que verlos como una forma de pasar dos argumentos al método:

- el caso base, de tipo *B*
- la función de plegado, que toma un objeto de la lista (de tipo *A*) y el resultado del plegado anterior (de tipo *B*) y devuelve un valor de tipo *B*

La función de plegado se aplica a la lista de izquierda a derecha y de derecha a izquierda respectivamente. Por ejemplo, si suponemos una lista con cuatro elementos, las funciones realizan los siguientes plegados:

```
lista = (a, b, c, d)
lista.foldRight(z)(f) => f(a, f(b, f(c, f(d, z))))
lista.foldLeft(z)(f) => f(f(f(f(z, a), b), c), d)
```

Los tipos *A* y *B* pueden ser el mismo. Por ejemplo el siguiente plegado concatena todas las

cadenas de una lista, ya que pasamos como función de plegado la función con concatenación:

```
val lista1 = List("hola", "que" , "tal")
lista1.foldRight("-")((c1: String, c2: String) => {c1 + c2})
⇒ String = holaquetal
```

La llamada a `foldRight` se podría simplificar de la siguiente forma:

```
lista1.foldRight("-")(_+_)
```

Y el siguiente ejemplo muestra un ejemplo de plegado en el que los tipos *A* y *B* son distintos:

```
lista1.foldRight(0) ((cad,n) => {cad.length+n})
⇒ Int = 10
```

Podemos ver la diferencia entre `foldRight` y `foldLeft`:

```
lista1.foldRight("-")(_ + _) ⇒ String = holaquetal-
lista1.foldLeft("-)(_ + _) ⇒ String = -holaquetal
```

5. Ámbitos

5.1. Reglas de evaluación de expresiones Scala con ámbitos

El funcionamiento de los ámbitos en Scala es igual al de Scheme:

- Una invocación a una función crea un nuevo ámbito en el que se evalúa el cuerpo de la función. Es el *ámbito de evaluación* de la función.
- El ámbito de evaluación se crea dentro del ámbito en el que se definió la función a la que se invoca
- Los argumentos de la función son variables locales de este nuevo ambito que quedan ligadas a los parámetros que se utilizan en la llamada.
- En el nuevo ámbito se pueden definir variables locales.
- En el nuevo ámbito se pueden obtener el valor de variables del ámbito padre.

Primer ejemplo

Supongamos el siguiente código en Scala:

```
def f(x: Int, y: Int): Int = {
        val z = 5
        x+y+z
```

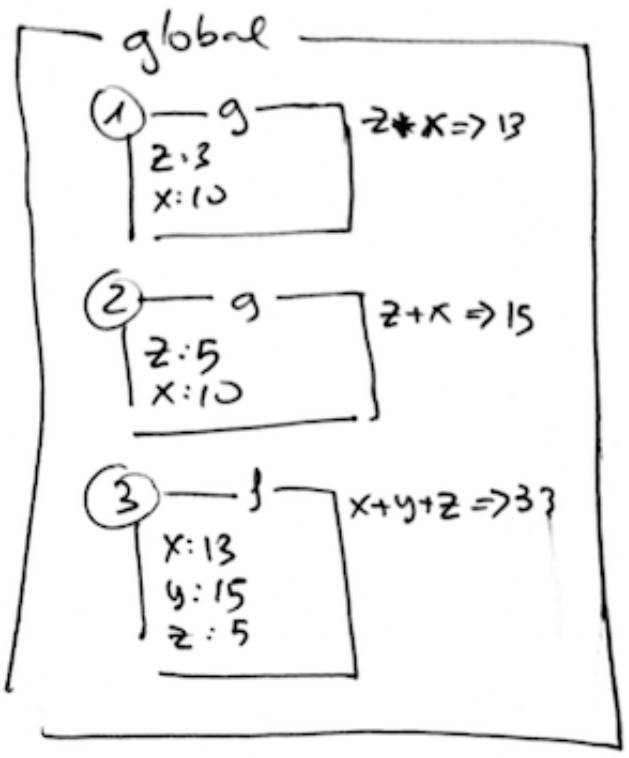
```
}
def g(z: Int): Int = {
        val x = 10
        z+x
}
f(g(3),g(5))
```

Se definen dos funciones `f` y `g` que definen cada una distintas variables locales y devuelven una suma de los parámetros con la variable local. En la última línea de código se realiza una invocación a `f` con los resultados devueltos por dos invocaciones a `g`.

¿Cuántos ámbitos locales se crean? ¿En qué orden?

1. En primer lugar se realizan las invocaciones a `g`. Cada una crea un ámbito local en el que se evalúa la función. Las invocaciones devuelven 13 y 15 respectivamente.
2. Después se realiza la invocación a `f` con esos valores 13 y 15. Esta invocación vuelve a crear un ámbito local en el que se evalúa la expresión `++y+z`, devolviendo 33.

El diagrama de ámbitos locales generados por las sentencias anteriores es el siguiente:



Segundo ejemplo con variables locales y globales

Supongamos el siguiente código en Scala:

```
val x = 10
val y = 20

def g(y: Int): Int = {
```

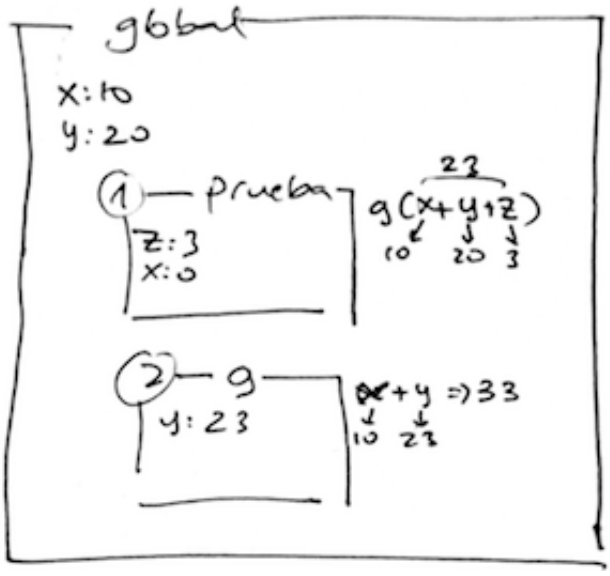
```
        x+y
}

def prueba(z: Int): Int = {
   val x = 0
   g(x+y+z)
}

prueba(3)
```

¿Qué devuelve `prueba(3)` ? ¿En qué ámbito se evalúa la expresión `x+y+z` ? ¿Y la expresión `x+y` ? ¿Qué valores tienen esas variables en el momento de la evaluación? Tenemos que aplicar el modelo de ámbitos de Scala para descubrirlo.

En el caso del ejemplo, si ejecutamos el código en el intérprete veremos que devuelve 33. El funcionamiento es igual a Scheme:

1. En el ámbito principal del intérprete se define el valor de las variables `x` (10) e `y` (20), así como la función `prueba` .
2. La ejecución de `prueba(3)` crea un nuevo ámbito de evaluación, dentro del ámbito principal, en el que se ejecuta el cuerpo de la función.
3. En el ámbito de evaluación se liga el valor de `z` (argumento de `prueba`) con el valor 3 (parámetro con el que se realiza la llamada).
4. En este nuevo ámbito se ejecuta el cuerpo de la función: se crea la variable local `x` con el valor `0` y se evalúa la expresión `x+y+z` . El valor de `x` y `z` están definidos en el propio ámbito local (0 y 3). El valor de `y` se obtiene del entorno padre: 20. El resultado de la expresión es 23.
5. Se invoca a la función `g` con el valor 23 como parámetro `y` . Para evaluar esta invocación se crea un nuevo ámbito local en el ámbito global en donde está definida la función.
6. Dentro de este nuevo ámbito se evalúa la expresión `x+y` devolviéndose 33 como resultado.

El diagrama de ámbitos que representa las invocaciones anteriores es el siguiente:



Por ahora es bastante normal. La diversión empieza cuando utilizamos la característica de que Scala (al igual que Scheme) puede construir funciones anónimas en la ejecución de otras funciones. Esto nos permite la definición de clausuras.

5.2. Clausuras en Scala

Siguiendo con el tema del funcionamiento de los ámbitos, ¿qué pasaría en el ejemplo anterior si `prueba` no evualúa ninguna expresión sino que devuelve una función anónima? Veámoslo, con una versión algo distinta del ejemplo anterior:

```
val x = 10
val y = 20
def prueba(y: Int): (Int)=>Int = {
   val x = 0
   (z:Int) => {x+y+z}
}
```

Hemos cambiado la definición de `prueba` haciendo que tome como parámetro la variable `y` y que devuelva una función anónima construida en tiempo de ejecución: la función `(z:Int) => {x+y+z}` que tiene como parámetro `z` y como variables libres `x` e `y`. La función es una clausura que se creará en al ámbito de evaluación de `prueba` y quedará asociada a ese ámbito en el momento en que se ejecute `prueba`.

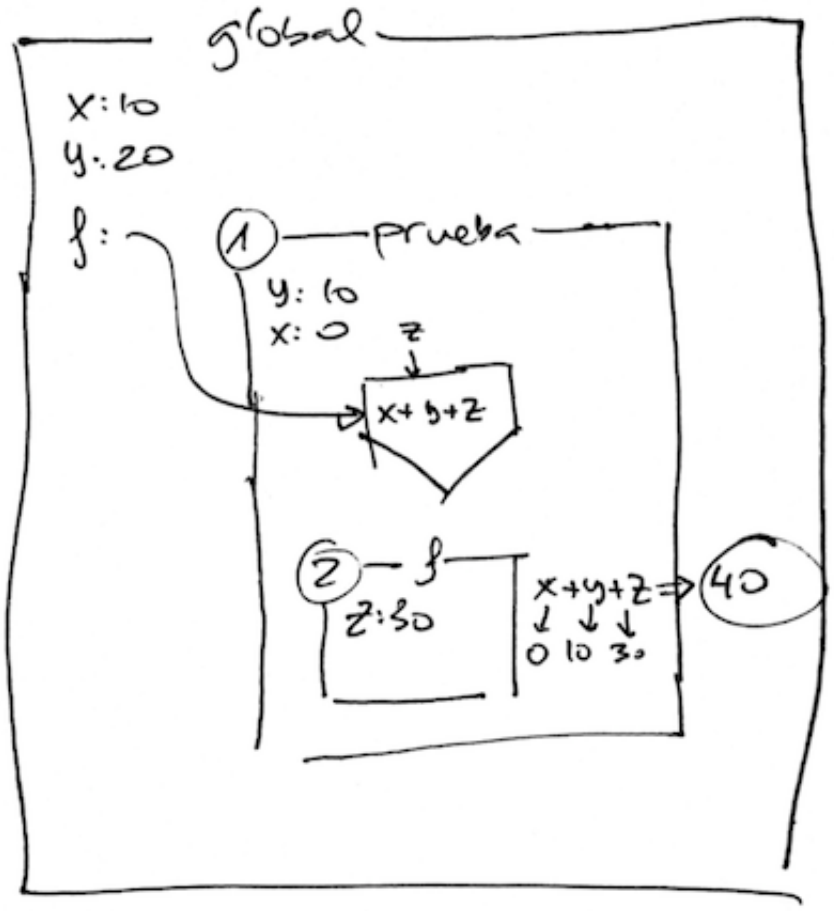
Supongamos que ejecutamos el siguiente código:

```
val f: (Int) => Int = prueba(10)
f(30)
```

Aunque no es necesario indicar el tipo de `f` (el compilador de Scala lo infiere) lo hacemos en el ejemplo, para que se entienda mejor qué devuelve `prueba`. El ejemplo funciona de la siguiente forma:

1. Se invoca `prueba(10)` y se crea su ámbito de evaluación (dentro del ámbito global) en el que se definen las variables locales `x` e `y` con los valores 0 y 10.
2. En este ámbito se crea la función anónima `(z:Int) => {x+y+z}`. Se devuelve una referencia a esa función que se guarda en la variable `f`. La función anónima permanece en el ámbito de evaluación de prueba, junto con las variables locales. Es una clausura.
3. La invocación a `f` crea un nuevo ámbito de evaluación, dentro del de `prueba` y allí se evalua la expresión `x+y+z`. El valor de `z` es 30 (valor asignado al argumento en la llamada a `f`. El valor de `x` e `y` no está definido en el ámbito, pero sí en su padre: 0 y 10. El valor resultante de la expresión es 40.

El diagrama de ámbitos que ilustra este comportamiento es el siguiente:



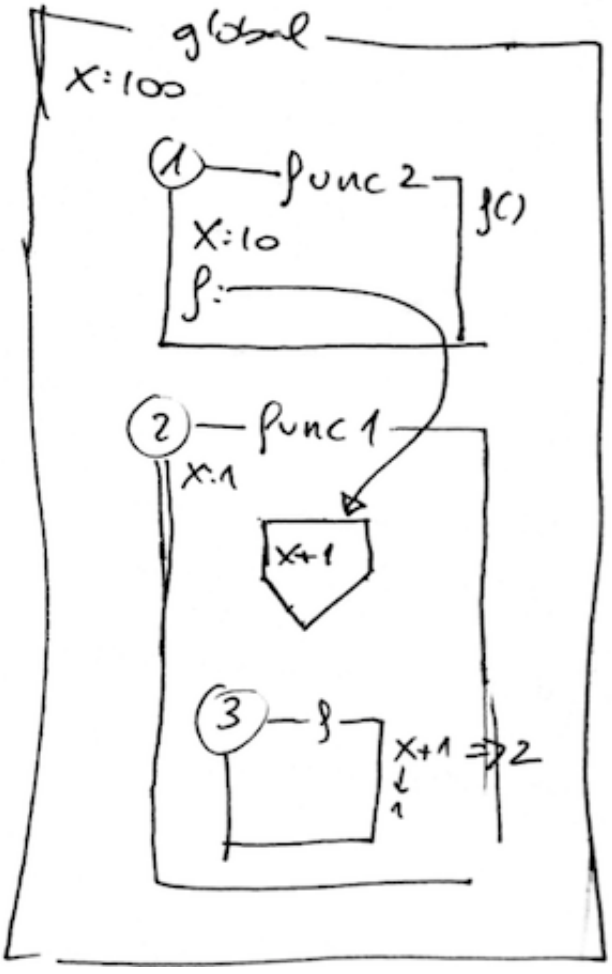
Algunos ejemplos de clausuras, similares a las que vimos en Scheme:

```
def makeSumador(k: Int): (Int)=>Int = (x: Int) => x + k
val f = makeSumador(10)
val g = makeSumador(100)
f(4)
g(4)
```

Y un último ejemplo

```
val x = 100
```

```
def func1(): () => Int = {
        val x=1
        () => {x+1}
}

def func2(): Int= {
        val x=10
        val f=func1()
        f()
}

func2()
```



La llamada a `func2` hace que se evalue su cuerpo. En él se crea la variable `x` con el valor 10 y la expresión `f=func1()` asocia `f` con la clausura devuelta por `func1`. Esta clausura es `()=>{x+1}` creada en el ámbito de evaluación de `func1`. En ese mismo ámbito se ha creado también la variable `x` con el valor 1. La llamada a `f` en el código equivalente hace que la función se evalue.

El ámbito de evaluación de `f` se crea en el ámbito de evaluación de `func1`, en el que `x` vale 1. El resultado de la llamada será 2.

Una formulación equivalente de `func2` es:

```
def func2(): Int= {
        val x=10
        func1()()
}
```

La llamada a `func1()` devuelve otra función que es evaluada con los siguientes paréntesis.

El código equivalente en Scheme sería:

```
(define x 100)

(define (func1)
   (let ((x 1))
       (lambda () (+ x 1))))

(define (func2)
   (let ((x 10))
      ((func1))))

(func2)
```

6. Otras características funcionales

6.1. Definición por comprensión de colecciones

Matemáticamente, una definición por comprensión de un conjunto se realiza con una definición lógica de una propiedad que deben satisfacer sus elementos. Por ejemplo, veamos la siguiente definición:

```
S = { x | x ∈ N : x == x² }
```

La definición se lee de la siguiente forma:

> S es el conjunto de números con dominio en los números naturales (N) que tienen la propiedad de que el número coincide con el número al cuadrado.

El conjunto de números que cumple esta propiedad es {0, 1}.

Otro ejemplo es el conjunto de los números pares, que podríamos definir de la siguiente forma:

```
Pares = { x | x ∈ N : x % 2 == 0 }
```

Muchos lenguajes de programación funcional permiten definir listas utilizando propiedades similares. Las listas tienen que tener un número finito de elementos, por lo que no podemos hacer una definición tan general como la anterior. En su lugar podemos hacer definiciones como la siguiente:

```
{ x² | x ∈ {1..5} }
```

Estamos definiendo una lista con los cinco primeros números naturales elevados al cuadrado. Es muy sencillo expresar esta lista en Scala:

```
for (x <- (1 to 5)) yield x*x
⇒ Vector(1, 4, 9, 16, 25)
```

La expresión `(1 to 5)` devuelve un *rango* de números del 1 al 5:

```
(1 to 5)
⇒ Range(1, 2, 3, 4, 5)
```

Un rango es un tipo particular de colección de Scala. La sentencia `for` ... `yield` aplica la función definida tras el `yield` a todos los elementos de la colección sobre la que itera `x` y devuelve otra colección transformada con la expresión. En este caso, un vector con los números 1 al 5 elevados al cuadrado.

El operador `yield` también se puede aplicar a listas. En este caso devuelve una lista en lugar de un vector. Ambos tipos de datos son colecciones.

```
for (x <- List.range(1,6)) yield x*x
⇒ List(1, 4, 9, 16, 25)
```

Es posible también definir una condición que deben cumplir los números, antes de aplicar la expresión que va generando la lista final. Por ejemplo, la siguiente expresión devuelve el cuadrado de los números impares del 1 al 100:

```
for (i <- List.range(1, 30) if i % 2 != 0) yield i*i
⇒ List[Int] = List(1, 9, 25, 49, 81, 121, 169, 225, 289, 361, 441, 529, 625, 729,
```

También es posible definir condiciones más complicadas. Por ejemplo, la siguiente expresión devuelve el cuadrado de los números impares divisibles por 3:

```
for (i <- List.range(1, 101) if (i % 2 != 0 && i % 3 == 0)) yield i*i
⇒List[Int] = List(9, 81, 225, 441, 729)
```

Y también es posible anidar varios `for` recorriendo más de una colección para tomar los distintos elementos que generarán la colección final. En el siguiente ejemplo se devuelve una colección de parejas formadas con dos generadores. El primero `(1 to 3)` define los primeros números de las parejas y el segundo `(1 to x)` utiliza la variable del primer generador como tope para obtener los valores de los segundos números:

```
for(x <- (1 to 3 ); y <- (1 to x)) yield (x,y)
⇒ Vector((1,1), (2,1), (2,2), (3,1), (3,2), (3,3))
```

Una nota muy importante: aunque el bucle `for` es tipicamente procedural, en las definiciones anteriores por comprensión de Scala se utiliza de forma funcional. La sentencia no tiene efectos laterales y devuelve una colección. Es posible implementar sentencias similares de forma recursiva.

6.2. *Currying*

Currying es el proceso de convertir una función que toma *n* argumentos en funciones que toman un argumento. Cada función devuelve otra función que consume el segundo argumento.

Veamos un ejemplo. La siguiente función toma un argumento `x`, devolviendo a su vez otra función que toma otro argumento `y` y que evalúa la expresión `x * y`:

```
def mulUnoCadaVez(x: Int) = (y: Int) => x * y
```

Para multiplicar dos números, se puede invocar la función de la siguiente forma:

```
mulUnoCadaVez(6)(7)
⇒ Int = 42
```

El resultado de `mulUnoCadaVez(6)` es la clausura `(y: Int) => x * y` con el valor `x` asociado a 6. Esta función se aplica a 7, devolviendo 42.

En Scala existe una forma abreviada de definir estas funciones *currificadas*:

```
def mulUnoCadaVez(x: Int)(y: Int) = x * y;
```

Bibliografía

- Martin Odersky: “Programming in Scala”
- Cay S. Horstmann: “Scala for the impatient”, Addison-Wesley Professional, 2012.
- Martin Odersky, “A Postfunctional Language”
- Mario Gleichman, Serie de posts “Functional Scala”

Lenguajes y Paradigmas de Programación, curso 2014–15

Domingo Gallardo, Cristina Pomares