Nuevas Tecnologías de la Programación

Tema 5: funciones

Curso 2017-18

Índice

Introducción

Las funciones son el elemento clave en programación funcional. En realidad este paradigma se basa en la creación de funciones reutilizables y que se pueden componer haciendo que sobre ellas pueda articularse el código.

Las sentencias se basarán en combinar funciones básicas generando cadenas de operaciones: la salida de una función se usa como entrada para otra llamada y así sucesivamente.

Introducción

Podemos decir que las funciones son expresiones con nombre, de forma que pueden reutilzarse. La programación en **Scala** (en los lenguajes de programación funcional) en la creación de funciones puras que:

- reciben valores como argumentos
- realizan cálculos sobre ellos
- devuelven un valor (que depende únicamente de los argumentos de entrada)
- no usan ni afectan datos externos a la función

Introducción

Las funciones puras son equivalentes a las funciones matemáticas, donde la definición es un cálculo derivado únicamente del valor de entrada.

Al final el código debe afectar a datos externos (escribiendo archivos, mostrando información por pantalla, almacenando datos en bases de datos, etc). Pero el objetivo de la programación funcional es reducir el número de funciones no puras.

Introducción

La sintaxis de declaración de una función es:

```
def <identificador> = <expresion>
```

Como ejemplo:

```
scala> def saludo="Hola"
saludo: String

scala> saludo
res0: String = Hola
```

Introducción

Una forma de declaración más completa incluye el tipo de devolución:

```
def <identificador> : <tipo> = <expresion>
```

```
scala> def saludo:String="Hola"
saludo: String
```

Introducción

En el caso de disponer de argumentos:

```
def <identificador>(<identificador>:tipo [,...]) : <tipo> =
                    <expresion>
```

Ejemplo:

```
scala> def multiplicar(x:Int, y:Int):Int = {x*y}
multiplicar: (x: Int, y: Int)Int

scala> multiplicar(10,17)
res0: Int = 170
```

Introducción

El cuerpo de las funciones consiste esencialmente en expresiones o bloques de expresiones, donde la línea final se convierte en el valor devuelto por la función.

La sentencia return se usará únicamente en aquellas situaciones en que se desee forzar la salida de una función:

```
def quitarBlancosIniciales(s:String) : String = {
   if(s !=null) return null
   s.trim()
}
```

Índice

Procedimientos

Se denomina así en **Scala** a aquellas aquellas funciones que no devuelven nada. Por ejemplo, funciones que finalizan mostrando algún mensaje por pantalla. En este caso, **Scala** infiere que el tipo de retorno es Unit.

Conviene indicar de forma explícita este tipo de retorno para que quede claro que se trata de un procedimiento y no de una función.

Procedimientos

```
scala> def mostrar(d:Double) : Unit =
                        println(f"Valor de $d%.2f")
mostrar: (d: Double)Unit

scala> mostrar(8.34567)
Valor de 8,35
```

Índice

Funciones sin argumentos

En el caso de funciones sin argumentos puede realizarse la llamada sin usar los paréntesis, como se ha visto antes en la función saludo.

Por convenio, se recomienda definir funciones sin argumentos con paréntesis si tienen efectos laterales (es decir, si no son funciones puras). Por ejemplo, si se encargan de mostrar algún mensaje por pantalla.

Índice

Llamadas a funciones con bloques de expresiones

En funciones con un único argumento es posible realizar la llamada mediante un bloque de expresiones (o expresión). Esto permite obtener el valor del parámetro actual a partir de algún cálculo previo, sin necesidad de crear alguna variable intermedia para almacenar el valor.

```
scala> def formatear(valor:Double) = f"$valor%.2feu"
formatear: (valor: Double)String

scala> formatear(2.98)
res4: String = 2,98eu

scala> formatear{23*0.15+7}
res5: String = 10,45eu
```

Índice

Funciones recursivas

En este tipo de funciones es imprescindible indicar el tipo de retorno:

```
scala> def factorial(x:Int) : Int = {
     | if(x == 0) 1
     | else x*factorial(x-1)
     | }
factorial: (x: Int)Int

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

Funciones recursivas

Siempre que sea posible interesa hacer que la función sea **tail-recursive**. En este caso es posible optimizar la ejecución de estas funciones evitando el uso de la pila. Para conseguirlo es necesario que la última sentencia sea la nueva llamada recursiva, exclusivamente. En otro caso no es posible realizar la optimización.

Funciones recursivas

Es posible agregar una anotación a la función recursiva, de forma que se comprueba si la función puede optimizarse o no. Si se agrega esta anotación a la función anterior se mostrariía un error, ya que tras la llamada recursiva es necesaria una multiplicación.

```
scala> @annotation.tailrec
     | def factorial(x : Int) : Int = {
     |    if(x == 0) 1
     |    else x*factorial(x-1)
     | }
<console>:14: error: could not optimize @tailrec annotated
    method factorial: it contains a recursive call not in
    tail position
         else x*factorial(x-1)
                  ^
```

Funciones recursivas

Una forma de resolver este problema consiste en ir acumulando el resultado sobre un argumento:

```
scala> @annotation.tailrec
     | def factorial(x : Int, acum : Int) : Int = {
     |    if(x == 0 || x == 1) acum
     |    else factorial(x-1, x*acum)
     | }
factorial: (x: Int, acum: Int)Int

scala> factorial(25,1)
res3: Int = 2076180480
```

Funciones recursivas

Otra forma manteniendo la interfaz natural de la función:

```
scala> def factorial(x : Int) : Int = {
     |     @annotation.tailrec
     |     def iterar(x : Int, acum : Int) : Int = {
     |         if(x == 0 || x == 1) acum
     |         else iterar(x-1, x*acum)
     |     }
     |     iterar(x,1)
     | }
factorial: (x: Int)Int

scala> factorial(10)
res0: Int = 3628800
```

Se usa una función auxiliar llamada iterar que sí que admite la optimización al ser tail recursive.

Índice

Parámetros con valor por defecto y llamadas con parámetros por nombre

Una forma de evitar el problema de ese argumento adicional en la primera versión del factorial, como forma de conseguir la optimización, consiste en asignar valor por defecto al argumento, de forma que no hay que especificar valor para él en la mayoría de los casos:

```
scala> scala> def factorial2(x : BigInt, acum:BigInt=1) : BigInt = {
     |     if(x == 0 || x == 1) acum
     |     else factorial2(x-1, x*acum)
     | }
factorial2: (x: BigInt, acum: BigInt)BigInt

scala> factorial2(30)
res7: BigInt = 265252859812191058636308480000000
```

Parámetros con valor por defecto y llamadas con parámetros por nombre

En caso de no especificar valor para el segundo argumento, se asume que es válido el valor por defecto. Con esto la llamada puede hacerse de la forma natural, pasando como argumento únicamente el valor del que se desea calcular el factorial.

Habitualmente los parámetros actuales se hacen corresponder 1 a 1 con los parámetros fomales. En **Scala** podemos hacer que las llamadas se realicen especificando los nombres de los parámetros formales, lo que permite alterar el orden establecido en la declaración.

Parámetros con valor por defecto y llamadas con parámetros por nombre

Por ejemplo, la llamada a factorial2 podría haberse hecho;

```
scala> factorial2(acum=1, x=30)
res8: BigInt = 265252859812191058636308480000000
```

Índice

Funciones anidadas

En **Scala** se permite el anidamiento de funciones, como hemos visto en el ejemplo anterior de la función auxuliar dentro de factorial2. Tiene sentido en situaciones en que debe evitarse la repetición de código, pero considerando que esta repetición se produce únicamente en el ámbito de la función.

```
scala> def max(a:Int, b:Int, c:Int) = {
     |     def max(x:Int, y:Int) = {
     |         if(x > y) x else y
     |     }
     |     max(a, max(b,c))
     | }
max: (a: Int, b: Int, c: Int)Int

scala> max(97,23,560)
res10: Int = 560
```

Índice

Número variable de argumentos, grupos y genéricos

En **Scala** es posible definir funciones con listas de argumentos de tamaño variable, con la única limitación de no poder haber argumentos *normales* tras la lista de tamaño variable (ya que no habría forma de distinguirlos). En el cuerpo de la función estos argumentos pueden usarse como iteradores en expresiones for. Una lista de argumentos de tamaño variable se denota mediante el carácter *:

```
scala> def sumar(numeros:Int *) : Int = {
     |     var total=0
     |     for(i <- numeros) total+=i
     |     total
     | }
sumar: (numeros: Int*)Int
```

Número variable de argumentos, grupos y genéricos

```
scala> sumar(1)
res12: Int = 1

scala> sumar(1,2,3)
res13: Int = 6

scala> sumar(1,2,3,4,5,6,7)
res14: Int = 28
```

Número variable de argumentos, grupos y genéricos

También es habitual que una función cuente con varias listas separadas de argumentos:

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

scala> max(3)(12)
res15: Int = 12

val max3=max(3)_
max3: Int => Int = $$Lambda$1184/2125274496@1d1bf7bf

scala> max3(15)
res18: Int = 15
```

Número variable de argumentos, grupos y genéricos

La ventaja de contar con varias listas de argumentos se pondrá de manifiesto más adelante, cuando se considere el paso de funciones como argumentos.

El uso de genéricos puede hacer que una función sea más útil y flexible, haciendo que los tipos se fijen en el momento de la llamada y no en la declaración. Supongamos que deseamos implementar una función genérica que devuelva todos los elementos de una lista, excepto el primero. Lo ideal sería implementar una función que pudiese trabajar con listas de todo tipo

Número variable de argumentos, grupos y genéricos

```
scala> def eliminarPrimero[A](lista: List[A]) = lista.tail
eliminarPrimero: [A](lista: List[A])List[A]

scala> eliminarPrimero(List(1,2,3,4))
res19: List[Int] = List(2, 3, 4)

scala> eliminarPrimero(List('a','b','c'))
res20: List[Char] = List(b, c)

scala> eliminarPrimero(List("hola", "adios"))
res21: List[String] = List(adios)
```

Índice

Métodos y operadores

Un método es una función declarada en el ámbito de una clase. Se llama de la forma habitual en orientación a objetos.

```
scala> eliminarPrimero(List("hola", "adios"))
res21: List[String] = List(adios)

scala> val cadena="Hola, mundo"
cadena: String = Hola, mundo

scala> val terminaEnO=cadena.endsWith("o")
terminaEnO: Boolean = true
```

Métodos y operadores

Los métodos existen en todos los tipos ofrecidos por **Scala** (recordemos que no hay tipos primitivos):

```
fscala> val d=7.87
d: Double = 7.87

scala> d.round
res22: Long = 8

scala> d.floor
res23: Double = 7.0
```

Métodos y operadores

```
scala> d.compare(17)
res25: Int = -1

scala> d.+(3.5)
res26: Double = 11.370000000000001
```

Podemos llamar a los métodos con notación de operador: $2 + 3$ equivale realmente a $2.+(3)$:

```
scala> d compare 13
res27: Int = -1

scala> d + 2.4
res28: Double = 10.27
```

Índice

Funciones de primera clase: tipos de funciones

El término función de primera clase alude a que las funciones son un tipo más en el lenguaje de programación, por lo que pueden crearse literales tipo función, almacenarse en variables o estructuras de datos, pasarse como argumento a otras funciones o ser a su vez el resultado producido por una función.

Las funciones que reciben otras funciones como argumento se denominan funciones de orden superior (higher-order functions). Ya hemos manejado alguna de ellas, como map, reduce o filter. La ventaja que ofrecen es poder crear funciones genéricas, donde algunos detalles de procesamiento se pasan a las funciones como argumentos.

Funciones de primera clase: tipos de funciones

El tipo de una función es un agrupamiento de :

- tipos de argumentos
- tipo de salida
- los tipos se conectan mediante el símbolo $=>$ (indica la dirección desde los tipos de entrada al tipo de salida)

```
scala> def duplicar(x:Int):Int=x*2
duplicar: (x: Int)Int
```

Funciones de primera clase: tipos de funciones

Al ser tratadas como objetos de cualquier otra clase es posible asignar funciones a variables.

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

scala> val funcionMaximo:(Int,Int)=>Int = buscarMaximo
funcionMaximo: (Int, Int) => Int =
               $$Lambda$1217/207366788@42a89cef

scala> val funcionMaximo2 = buscarMaximo _
funcionMaximo2: (Int, Int) => Int =
               $$Lambda$1218/308784574@6700374f
```

Funciones de primera clase: tipos de funciones

A observar el uso del marcador de posición (_) que se usa como comodín para futuras llamadas a la función. Indica que no se especifica ningún argumento y que deberán indicarse en el momento en que se produzca la llamada.

Índice

Funciones de orden superior

Como ya se ha indicado, se trata de funciones que reciben alguna otra función como argumento o devuelven una función como resultado. Veamos algunos ejemplos:

```
def operacionSeguraString(cadena : String,
                          operacion : String => String) = {
      if(cadena != null) operacion(cadena) else cadena
}
```

Se aprecia que la operación no se ha indicado de forma directa, sino que se pasa como argumento a la función (por lo que resulta muy versátil y general).

Funciones de orden superior

Podemos definir ahora varias operaciones sobre cadenas (candidadatas a ser pasadas como argumentos a la función anterior):

```
def invertir(cadena : String) = cadena.reverse
```

```
def aMayuscula(cadena : String) = cadena.toUpperCase
```

Funciones de orden superior

La forma de uso de la función genérica de aplicación sergura de funciones sobre cadenas se haría de la siguiente forma:

```
// Se usa la operacion segura
val res1 = operacionSeguraString("Hola", invertir)
println(s"Inversion sobre Hola: $res1")

val res2 = operacionSeguraString("Hola", aMayuscula)
println(s"Hola en mayuscula: $res2")

val res3 = operacionSeguraString(null, invertir)
println(s"Inversion sobre null: $res3")
```

Funciones de orden superior

Otro ejemplo interesante permite realizar la composición de funciones: $f(g(x))$.

```
def componer(f: Double => Double,
             g: Double => Double) : Double => Double =
      x => f(g(x))

// Se considera alguna aplicacion
def mas5(x:Double) = x+5
def cuadrado(x:Double)= x*x

// Se obtiene la funcion compuesta
val fg=componer(mas5, cuadrado)

// Se usa la funcion compuesta
val valor=fg(5)
println("Aplicacion de fg sobre 5: "+valor)
```

Índice

Literales tipo función

Se trata de funciones anónimas (carecen de nombres) y que pueden asignarse a valorses tipo función. Por ejemplo:

```
scala> val multiplicarPor2 = (x:Int) => x*2
multiplicarPor2: Int => Int = $$Lambda$1011/2095373876@67b7c170
```

El literal tipo función es la expresión a la derecha del igual (también se denomina expresión lambda o función anónima). Por su parte los valores (o variables) tipo función son variables a las que se les asigna una función.

Literales tipo función

La asignación a una variable tipo función puede hacerse de diferentes formas:

```
// Declaracion de variable tipo funcion
val multiplicarPor2 = (x:Int) => x*2

// Declaracion de funcion
def calcularMaximo(a:Int, b:Int) = if (a > b) a else b

// Se asigna la funcion a una variable
val variableFuncion1 : (Int, Int) => Int = calcularMaximo

// La asignacion se hace con expresion lambda
val variableFuncion2 = (a:Int, b:Int) => if(a > b) a else b
```

Literales tipo función

También pueden asignarse literales de función sin argumentos:

```
// Declaracion de literal sin argumentos
val saluda = () => "Hola, mundo"

// Se debe de llamar con () para que se interprete
// como llamada
println(saluda())
```

Índice

Sintaxis del marcador de posición

Forma abreviada de usar las expresiones lambda, sustituyendo los parámetros con el carácter subrayado. Puede usarse si:

- se especifica el tipo de la función de forma explícita
- los parámetros se usan una única vez

Se consideran algunos ejemplos.

Sintaxis del marcador de posicion

```
val multiplicarPor2 : Int => Int = _ * 2
```

El uso es correcto ya que está indicado el tipo de la expresión lambda y el parámetro sólo se usa una vez, ya que la expresión equivale a:

```
x => x*2
```

Sintaxis del marcador de posicion

```
def operacionSeguraString(cadena : String,
                          operacion : String => String) = {
      if(cadena != null) operacion(cadena) else cadena
}

// Puede simplificarse usando el subrayado
operacionSeguraString("Hola", _.reverse)
```

Aquí la expresión con subrayado equivale a:

```
x => x.reverse
```

Sintaxis del marcador de posicion

También puede usarse en situaciones de más de un argumento:

```
def combinacion(x:Int, y:Int, f:(Int,Int) => Int) = f(x,y)

// Se usa la funcion previa
val resultado2=combinacion(12, 3, _*_)
```

Y la expresión usada para la función equivale a:

```
(x,y) => x*y
```

Esta sintaxis es especialmente útil al trabajar con colecciones y estructuras de datos, que ofrecen métodos que esperan recibir funciones como argumentos y así se simplifica el código necesario para usarlos.

Índice

Funciones parcialmente aplicadas y currying

Usualmente la llamada a una función se realiza usando todos los argumentos. Pero a veces puede resultar conveniente reutilizar una llamada a una función para retener el valor de algunos parámetros. Como ejemplo consideramos ejemplo de función con dos parámetros que comprueba si un número es divisible por otro.

```
def divisible(x:Int, y:Int) = x%y == 0
```

Podemos obtener una versión simplificada sin indicar ningún parámetro:

```
val f = divisible _

// Forma de uso
val resultado1=f(20,7)
```

Funciones parcialmente aplicadas y currying

También podemos retener el valor de alguno de los parámetros (lo que se conoce como función parcialmente aplicada), usando el operador subrayado, indicando de forma explícita el tipo del argumento:

```
val divisiblePor3 = divisible(_:Int, 3)

// Se usa con un unico argumento
val resultado2=divisiblePor3(27)
```

Funciones parcialmente aplicadas y currying

Una forma más cómoda de hacer esta asignación parcial de argumentos consiste en usar varias listas de argumentos (currying):

```
def divisiblePor(x:Int)(y:Int) = x%y == 0

// Forma de uso
val resultado3=divisiblePor(8)(2)
println(resultado3)
```

Funciones parcialmente aplicadas y currying

Ahora es más natural asignar alguno de ellos y mantener el resto libres:

```
val divisiblePor2 = divisiblePor (_:Int)(2)

// Y puede usarse de forma normal
val resultado4=divisiblePor2(100)
println(resultado4)
```

Una función con múltiples listas de argumentos se considera como una cadena de varias funciones. Cada lista es en realidad una llamada a función.

```
// version de una lista
(Int, Int) => Int

// version de dos lista
Int => Int => Boolean
```

Índice

Funciones parciales

Todas lasfunciones vistas hasta ahora son funciones totales: son capaces de tratar con todos los posibles valores de los argumentos. Sin embargo hay algunas funciones que no soportan todos los valores de entrada. Por ejemplo, la función que calcula la raíz cuadrada no puede calcular sobre valores negativos. En **Scala** es habitual tener funciones parciales asociadas al uso de la expresión match:

```
val gestorErrores : Int => String = {
      case 200 => "Funcionamiento correcto"
      case 400 => "Error tipo 1"
      case 500 => "Error tipo 2"
}

// Se genera error al ejecutar
val resultado=gestorErrores(450)
```

Índice

Funciones parciales

Es posible usar literales tipo función en llamadas a funciones:

```
def operacionSeguraString(s:String, f:String => String) = {
   if(s != null) f(s)
   else s
}

val resultado=operacionSeguraString("Hola, Pepe", {
   s => {
      val hora=System.currentTimeMillis()
      val cadenaFinal=s+"("+hora+")"
      cadenaFinal.toUpperCase
   }
})

println(resultado)
```

Índice

Documentación de funciones

El código debe ser legible para otros desarrolladores. Un código legible es también comprensible y fácil de mantener. Para conseguir este objetivo conviene seguir una serie de recomendaciones:

- las funciones no deben ser extensas (si es necesario pueden descomponerse en otras funciones auxiliares)
- los nombres deben estar bien elegidos e indicar su finalidad
- el código deben documentarse como en cualquier otro lenguaje de programación. La forma de documentación es muy parecida a la usada por **javadoc**