CURSO DE PROGRAMACION SCALA
Sesión 6

Sergio Couto Catoira

Índice

- Generalización de funciones de orden superior
- Funciones fold
- Parámetros por valor y por referencia
- Ejercicios, ejercicios, ejercicios

Funciones de orden superior

- Fíjate en los métodos sum y product definidos en la sesión anterior
- Hay apenas 3 cosas diferentes

```
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case Cons(x, xs) => x + sum(xs)
}

def product(ints: List[Double]) : Double = ints match {
  case Nil => 1.0
  case Cons(x, xs) => x * product(xs)
}
```

Funciones de orden superior

- Tipos de entrada / salida => ya sabemos como parametrizarlos
- Operación que realiza => ya sabemos como parametrizarlo
- Valor para caso especial => Podemos parametrizarlo del mismo modo

```
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case Cons(x, xs) => x + sum(xs)
}

def product(ints: List[Double]) : Double = ints match {
  case Nil => 1.0
  case Cons(x, xs) => x * product(xs)
}
```

Ejercicio

- Define una función **foldRight** que generalice las funciones sum y product
  - def foldRight[A, B] (as: Lista[A], z: B) (f: (A, B) => B) : B = ???

- Cosas que esta función lleva:
  - Parametrización de tipos: [A,B]
  - Valor por defecto: z
  - Función f separda en otro argumento para inferencia de tipos

Ejercicio

- Define las funciones sum y product en base a foldRight

Parámetros por valor o referencia

- En Scala los parámetros puedes pasarse por valor o por referencia
- Llamada por valor*(by-value)*
  - Evalúa el argumento una única vez (exactly once)
  - def callByValue(x: Int) = ???
- Llamada por referencia *(by-name)*
  - Se evalúa todas y cada una de las veces que se usa
  - Si no se usa, no se evalúa
  - def callByName(x: => Int) = ???

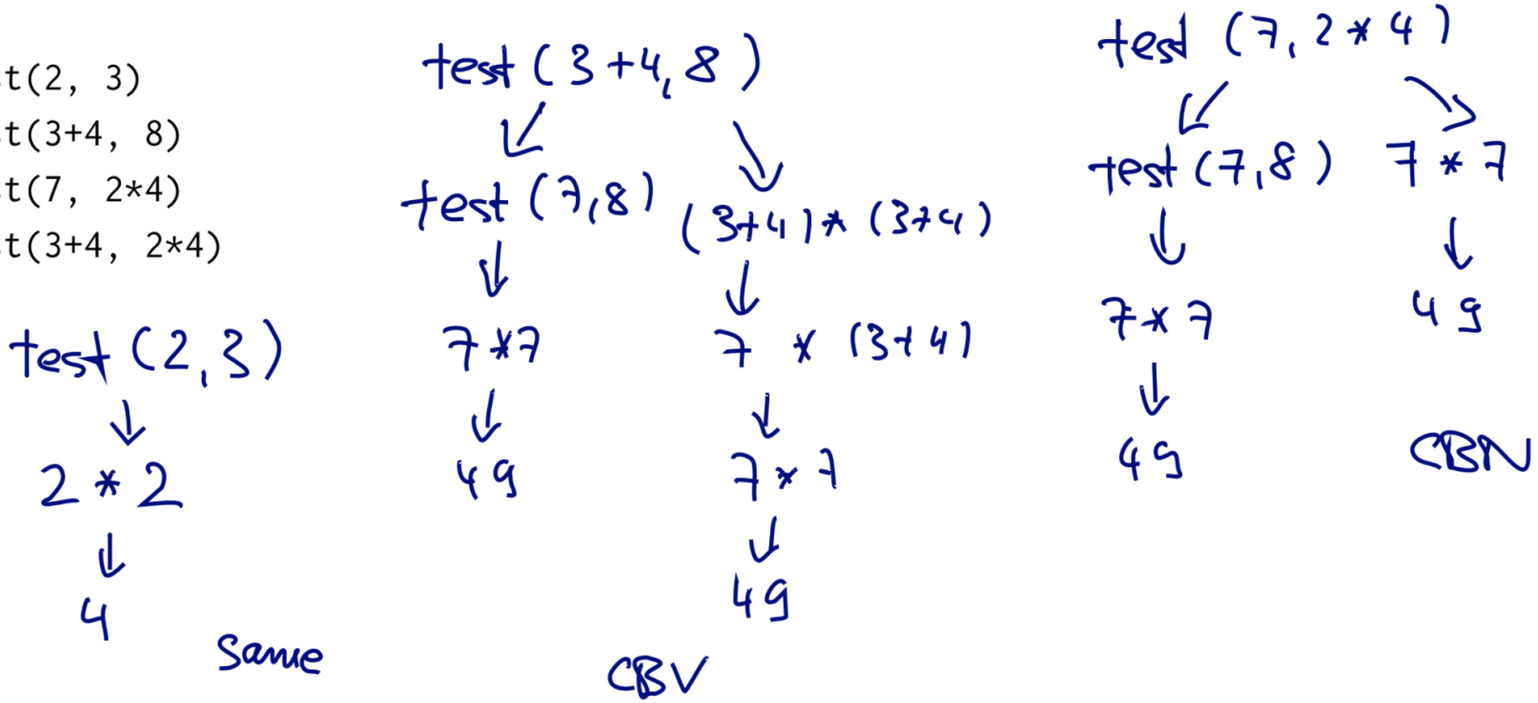
Parámetros por valor o referencia

- Según el caso, merece la pena uno u otro

Call-by-name vs call-by-value

```
def test(x: Int, y: Int) = x * x

test(2, 3)
test(3+4, 8)
test(7, 2*4)
test(3+4, 2*4)
```



Ejercicios

- Define la función length en base a foldRight

- La función foldRight no es tail-safe. Porque “acumula por la derecha”. Implementa una función foldLeft que sea tail-safe
  - def foldLeft[A, B] (as: Lista[A], z: B) (f: (B, A) => B) : B

FoldRight vs FoldLeft

FoldLeft: Asociativa desde la izquierda

(((1+2) +3) +0)

- foldLeft(Lista(1,2,3), 0)(_ +_)
- foldLeft(Lista(2,3), 1)(_ +_)
    - foldLeft(Lista(3), 3 (_ +_)
        - folfLeft(Vacio, 6) (_ + _)
            - 6

FoldRight: Asociativa desde la derecha

(1+(2 +(3 +0)))

- foldRight(Lista(1,2,3), 0)(_ +_)
- 1 + foldRight(Lista(2,3), 0)(_ +_)
    - 1 +2 + foldRight(Lista(3)(0)(_+ _)
        - 1+2+3 + foldRight(Vacio, 0) (_+_)
        - 1 + 2+ 3 + 0
            - 6

Ejercicios

- Define las funciones sum, product y length en base a foldLeft
- Define una función que devuelva una lista del revés.
  - def reverse[A](lista: Lista[A]) : Lista[A] = ???
  - Intenta hacerla usando un fold (pista: La función debe appendar listas, puedes probar con la función append definida anteriormente o con el constructr Cons)

Ejercicios

- Fijándote **bien** en las cabeceras de ambas funciones haz los siguientes ejercicios
  - Redefine foldLeft en base a foldRight
  - Redefine foldRight en base a foldLeft

Ejercicios

- Define la función append en base a foldLeft o foldRight (Pista: Usa el constructor Cons)
- Define una función que concatene lista de listas en una única lista.
  - Def appendLists(as: Lista[Lista[A]]
  - Pista: Usa la función anterior
  - Pista: Cuál debe ser el valor por defecto?