

Sergio Couto
s.couto@indizen.com



Introducción a Scala y Programación Funcional

Mayo 2018

ÍNDICE

- | | |
|-----------|--|
| 01 | Estructuras de datos funcionales |
| 02 | Funciones variádicas |
| 03 | Ejercicios sobre listas |
| 04 | Mejoras en inferencia de tipos |
| 05 | Data sharing |
| 06 | Generalización funciones de orden superior |

01 | Estructuras de datos funcionales

- > Sealed trait
 - Impide que se extienda un trait fuera del fichero donde está implementado
- > Case object
 - Representa caso base: vacío
- > Case class
 - Representan el resto de clases

- > Cons contiene dos elementos:
 - Head: primer elemento
 - Tail: resto de elementos (otra lista)

```
sealed trait Lista[+A]  
  
case object Vacio extends Lista[Nothing]  
  
case class Cons[+A](head: A, tail: Lista[A]) extends Lista[A]
```

>

02 | Funciones variádicas

- › Permiten recibir entre 0 y n parámetros separados por coma
 - `def apply(as: A *): Lista[A]`
- › Es azúcar sintáctico (syntactic sugar) para pasar secuencias de elementos.
- › Convertir una lista en secuencia: `_*`

(Sólo para pasarla como argumento a una función variádica)



03 | Ejercicios sobre listas - operaciones

- > Crea las funciones `sum` y `product` en el companion object `Lista` que aparece en el código. Recuerda usar pattern matching
 - `def sum(ints: Lista[Int]): Int = ???`
- > Crea una función `tail` que devuelva la cola de la lista.
 - `def tail[A](l: Lista[A]): Lista[A] = ???`
- > Crea una función `setHead`, que reciba una lista y una cabecera y reemplace la cabecera de la lista. Si la lista recibida es vacía, debe devolver una lista con un único elemento.
 - `def setHead[A](l: Lista[A], newHead: A): Lista[A] = ???`

03 | Ejercicios sobre listas - generalización

- > Generaliza la función tail en la función drop, que elimina n elementos de la lista. Recuerda hacerla tail safe
 - `def drop[A](l: Lista[A], n: Int): Lista[A] = ???`
- > Genera una función dropWhile que elimina los elementos de la lista mientras cumplan la función recibida. Debe recibir una lista y una función $A \Rightarrow \text{Boolean}$
 - `def dropWhile[A](l: Lista[A], f: A => Boolean): Lista[A] = ???`

04 | Mejoras en inferencia de tipos

- › Cuando se envía una función anónima a una función genérica, es necesario especificar el tipo de los parámetros
 - `dropWhile(lista, (x: Int) => x > 5)`
- › El primer parámetro, ya es una lista de enteros, por lo tanto es redundante indicarle el tipo de la función anónima
- › Scala permite una forma para que, currificando la función, sea capaz de inferir el tipo de la función anónima.
 - › `def dropWhile[A](l: Lista[A], f: A => Boolean)`
 - `dropWhile(lista, (x: Int) => x>5)`
 - › `def dropWhile[A](l: Lista[A])(f: A => Boolean)`
 - `dropWhile(lista)(x => x>5)`
 - `dropWhile(lista) (_ >5)`

- › Data sharing vs copia pesimista
 - Data sharing => Comparto objetos inmutables
 - Copia pesimista => Cada método hace una copia, por si algún otro método modifica algo
- › Todos los métodos que se generaron de momento, devuelven otra lista.
- › Sin embargo, en muchos casos devolvemos el mismo tail. No se hace copy, ni clone. **Es el mismo objeto.**
- › Es seguro porque es **immutable**.
- › Nos permite implementar funciones de forma eficiente tanto en tiempo como en memoria. No se hacen copias innecesarias de datos.



06 | Orden superior

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

- > 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` separada en otra lista de argumentos para inferencia de tipos

- > Define las funciones `sum` y `product` utilizando `foldRight`. Su signatura debe ser la misma que anteriormente.

- > Define una función `length` que usando `foldRight` devuelva la longitud de una lista

- > 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`
- > **FoldLeft**: Asociativa desde la izquierda. **Actualiza el caso base en cada iteración**
- > `((((1+2) +3) +0)`
 - `foldLeft(Lista(1,2,3), 0)(_ +_)`
 - `foldLeft(Lista(2,3), 1)(_ +_)`
 - `foldLeft(Lista(3), 3 (_ +_)`
 - `foldLeft(Vacio, 6) (_ +_)`
 - 6
- > **FoldRight**: Asociativa desde la derecha. Mantiene caso base. **Acumula resultado**
- > `(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

- > 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] = ???`

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

- > 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] = ???`

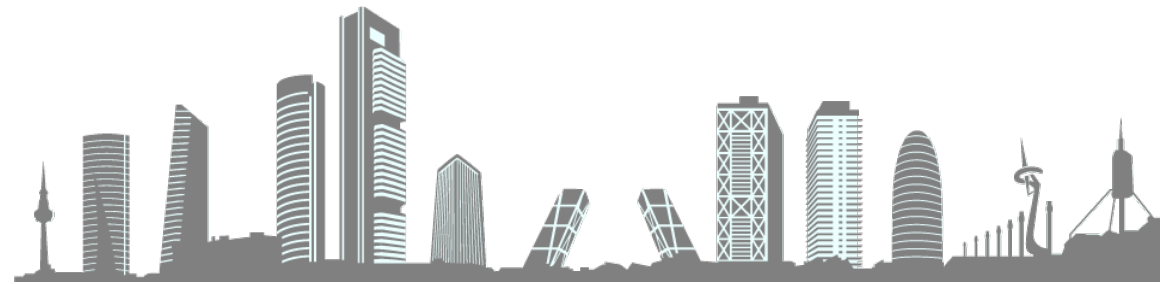
- > Fijándote **bien** en las cabeceras de ambas funciones haz los siguientes ejercicios
 - Redefine `foldLeft` en base a `foldRight`
 - Redefine `foldRight` en base a `foldLeft`

“Con
Indizen
nunca caminarás solo”



indizen[®]

Technology & Business Confluence



GRACIAS

www.indizen.com