

CURSO DE PROGRAMACION SCALA

Sesión 7

Sergio Couto Catoira

Índice

- > Funciones de *modificación y filtrado sobre TDA*
 - *Map*
 - *filter*
 - *FlatMap*
 - *zip*
- > Ejercicios, ejercicios, ejercicios

Ejercicios

- > Crea una función que, sobre una lista de enteros, añada 1 a cada elemento.
 - `def addOne(l: Lista[A]): Lista[A]`
- > Crea una función que, sobre una lista de Double, convierta cada elemento a String.
 - `def doubleToString(l: Lista[Int]): Lista[String]`
- > Intenta hacerlo empleando en ambos el método `foldRight` o `foldLeft`

Ejercicio – función map

- > Generaliza las funciones anteriores en una función map que aplique una función sobre cada elemento de la lista
 - `def addOne(l: Lista[A]): Lista[A]`

Ejercicio – función filter

- > Define una función filter que elimine todos los elementos que no cumplan la función
 - `def filter(l: Lista[Int])(f: Int => Boolean): Lista[Int]`

Ejercicio – función flatMap

- > Define una función flatMap que haga lo mismo que map pero devuelva una lista en lugar de un resultado sencillo
 - `def flatMap[A, B](l: Lista[A])(f: A => Lista[B]): Lista[B]`
 - Pista: En vez de fold, intenta usar alguna de las funciones anteriores.

Ejercicio – función filter

- > Implementa filter en base a flatMap.
 - Pista: Una función anónima que cree una lista para el elemento que cumpla f, y una lista vacía si no.

Ejercicio – función zip

- > Implementa una función que reciba dos listas de enteros y sume cada uno de los elementos:
 - `def addLists(l1: Lista[Int], l2: Lista[Int]): Lista[Int]`
- > Generaliza la función anterior para que admita cualquier tipo y cualquier función
 - `def zipWith[A,B,C](l1: Lista[A], l2: Lista[B])(f:(A,B) => C): Lista[C]`

Listas en la librería standard

- > $\text{Cons}(h, t) \Rightarrow h::t$
- > $\text{Cons}(h, \text{Cons}(h2, t)) \Rightarrow h::h2::t$
- > $\text{Vacio} \Rightarrow \text{Nil}$
- > Los métodos van incluidos dentro de la lista, por lo que la sintaxis sería:
 - Librería standard: `list.metodo(parametros)`
 - Nuestra librería: `metodo(list, parametros)`

Más métodos sobre listas

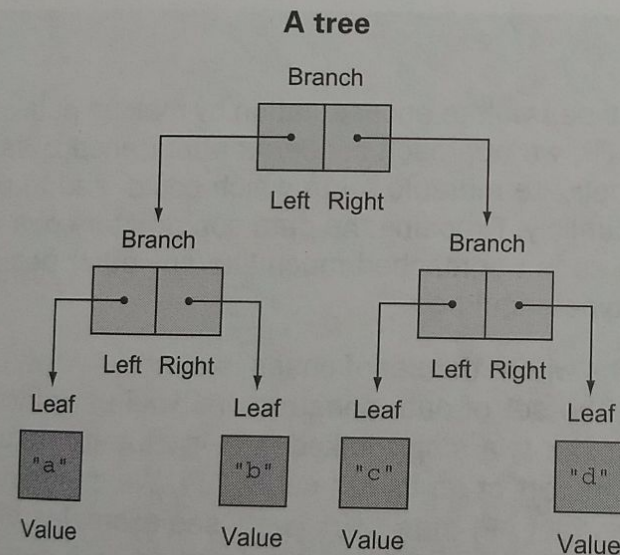
- > `take(n: Int)` Devuelve los primeros `n` elementos de la lista
- > `takeWhile(f: A => Boolean)` Devuelve elementos hasta que alguno incumpla la función
- > `forall(f: A => Boolean)` true si todos los elementos cumplen la función
- > `exists(f: A => Boolean)` true si algún elemento cumple la función
- > `scanLeft` && `scanRight`: Lo mismo que `fold`, pero devuelve lista con resultados parciales
 - `List(1,2,3).foldLeft(0)(_ + _) = 6`
 - `List(1,2,3).scanLeft(0)(_ + _) = List(0,1,3, 6)`

Otro TDA: Árboles

› Estructura de nodos

- Hoja => Valor sencillo
- Rama => Rupla de árboles

```
sealed trait Tree[+A]  
case class Leaf[A](value: A) extends Tree[A]  
case class Branch[A](left: Tree[A], right: Tree[A]) extends Tree[A]
```



Ejercicios sobre Árboles

- › Define la función `size` que devuelva el número de nodos (Hojas y ramas) de un árbol
 - `def size[A](t: Arbol[A]): Int`
- › Define la función `maximum` que devuelva el valor máximo de un árbol de enteros.
 - `def maximum(t: Arbol[Int]): Int`
- › Define la función `depth` que devuelva la profundidad máxima del árbol
 - `def depth[A](t: Arbol[A]): Int`

Más ejercicios sobre Árboles

- › Define la función `map` que reciba una función y devuelva un nuevo árbol con los nodos transformados
 - `def map[A, B](t: Arbol[A])(f: A => B): Arbol[B]`
- › Escribe una función `fold` que generalice las operaciones anteriores
 - `def fold[A, B](t: Arbol[A])(f: A => B)(g: (B, B) => B): B`
- › Redefine las funciones `size`, `maximum`, `depth` y `map` en base a `fold`

Ejercicio - difícil

- > Define una función subsecuencia que devuelva true si la lista contiene la subsecuencia
 - `tieneSubsecuencia[A](lista: Lista[A], sub: Lista[A]): Boolean`