




ASIGNATURA: PROGRAMACIÓN DECLARATIVA	 ACADEMIA UNIVERSITARIA Avenida Alcalde de Móstoles 33 (Posterior) 28922 Móstoles (Madrid) 91 664 67 20 - 717 716 540 info@academia-atika.com www.academia-atika.com
TITULACIÓN: INFORMÁTICA	
PROFESOR: JAVIER VERGARA IGUAL	
<h1>SCALA</h1>	

El paradigma de programación funcional está basado en el concepto de función pura. Una función pura es aquella que recibe unos argumentos de entrada y devuelve un valor, comportándose siempre de la misma forma, es decir, para los mismos argumentos siempre devolverá el mismo valor. El uso de funciones evita efectos laterales, es decir, la modificación del valor de variables dentro de un subprograma.

RECURSIVIDAD

La recursividad es esencial para resolver muchos ejercicios de listas, así como problemas de otras estructuras de datos. Existen 2 tipos de **recursividad** diferentes, **final** o **de cola**, o **no final**. La recursividad final o de cola se fundamenta en construir el resultado en un parámetro acumulador y, al llegar al caso base, devolverlo. Su esencia radica en que la llamada recursiva es la última instrucción que se ejecuta en una rama recursiva. Por el contrario, la recursividad no final no construye la solución en un parámetro acumulador, sino que el resultado de la propia llamada recursiva, se utiliza para dar forma al resultado. Por tanto, no es la última instrucción que se ejecuta en una rama recursiva. Centrándonos en la recursividad final o de cola, dado que necesitamos la función original y la auxiliar, que es la que realmente construye el resultado y contiene el algoritmo recursivo, podemos utilizar la notación **@annotation.tailrec**. Esta notación especial en Scala, le indica al compilador que debe optimizar el proceso recursivo, es decir, la gestión de la pila de memoria. Al parámetro acumulador se le suele llamar **acc**, aunque su nombre es indiferente. En la recursividad no final la lista resultado, si es que es necesario devolver una lista, se debe construir de derecha a izquierda para mantener el mismo orden que el original. En la recursividad final se construye de izquierda a derecha, es decir, en el mismo orden en el que se recorre. En resumen:

- **NO final:** construye la solución a partir del valor que devuelve el caso base. En caso de ser una lista, se construye de derecha a izquierda. La llamada recursiva no es la última instrucción que se ejecuta en una rama recursiva.
- **Final (de cola):** acumula la solución en un parámetro. Para ello, la función principal llama a una auxiliar que implementa el algoritmo recursivo y que, al llegar al caso base, devuelve el acumulado. En caso de ser una lista, se construye de izquierda a derecha.



ASIGNATURA: PROGRAMACIÓN DECLARATIVA	 ACADEMIA UNIVERSITARIA Avenida Alcalde de Móstoles 33 (Posterior) 28922 Móstoles (Madrid) 91 664 67 20 - 717 716 540  info@academia-atika.com www.academia-atika.com
TITULACIÓN: INFORMÁTICA	
PROFESOR: JAVIER VERGARA IGUAL	
<div>SCALA</div>	

AJUSTE DE PATRONES

El ajuste de patrones (*pattern matching*) permite analizar el valor de, por ejemplo, una lista o un árbol, mediante diferentes ramas *case*. Podemos encapsular en tuplas todos aquellos valores que queramos analizar. En una rama **case** de un **match** podemos, aparte analizar el valor, añadir una cláusula condicional mediante el uso de **if**, p.ej.:

```
@annotation.tailrec
def drop[A](list: List[A], n: Int): List[A] =
  list match {
    case _ :: tail if n > 0 =>
      drop(tail, n - 1)
    case _ =>
      list
  }
```



Las comillas francesas (``) sirven para representar a un identificador estable (*stable identifier*). Este identificador es un nombre que está vinculado estáticamente a un valor.

ASIGNATURA: PROGRAMACIÓN DECLARATIVA	 ACADEMIA UNIVERSITARIA Avenida Alcalde de Móstoles 33 (Posterior) 28922 Móstoles (Madrid) 91 664 67 20 - 717 716 540  info@academia-atika.com www.academia-atika.com
TITULACIÓN: INFORMÁTICA	
PROFESOR: JAVIER VERGARA IGUAL	
<div>SCALA</div>	

TUPLAS

Mediante el ajuste de patrones o la asignación a variables (**var**) o constantes (**val**), podemos descomponerlas en sus fragmentos. La sintaxis *aTuple._1*, *aTuple._2*, ... también nos permiten acceder a sus partes.

ATICA

ASIGNATURA: PROGRAMACIÓN DECLARATIVA	 <p>ACADEMIA UNIVERSITARIA</p> <p>Avenida Alcalde de Móstoles 33 (Posterior) 28922 Móstoles (Madrid)</p> <p>91 664 67 20 - 717 716 540 </p> <p>info@academia-atika.com www.academia-atika.com</p>
TITULACIÓN: INFORMÁTICA	
PROFESOR: JAVIER VERGARA IGUAL	
<h1>SCALA</h1>	



LISTAS

El tipo que representa a una lista es **List**. Esta clase tiene dos constructores disponibles: **Nil** y **List**. **Nil** representa a una lista vacía, pudiéndose también indicar como **List()**. Al utilizar ajuste de patrones, nos sirve cualquier representación. Para dividir la lista en el primero y el resto, usamos **head :: tail**. El constructor universal **List** se puede utilizar para especificar una lista con un número concreto de elemento, aunque el operador **::** también nos sirve para este propósito. En resumen:

- cabeza (**head**) :: cola (**tail**)
- **List**: Constructor Universal
- **Nil**: alias del constructor de una lista vacía **List()**
- **::** agrega un elemento por el principio de la lista, es decir, como cabeza
- **++** concatena 2 listas. También se puede usar el operador **:::**

FUNCIONES

- **aList.foldLeft(initial_value)(binary_operator_function)**: El método (función) *foldLeft* aplicado sobre una lista, tiene como objetivo recorrer todos y cada uno de los elementos de dicha lista de izquierda a derecha. Usando como punto de partida un valor inicial, que puede ser de cualquier tipo, pliega cada uno de los elementos utilizando una función de operador binario, es decir, que recibe dos parámetros. El primero de esos parámetros representa al valor inicial y se utiliza como acumulador. El segundo, es el elemento de la lista. *foldLeft* devuelve como resultado el valor asociado al parámetro acumulador después de haber recorrido todos los elementos de la lista, es decir, su tipo de retorno es el mismo que el del valor inicial. Si la lista está vacía, el resultado es el valor inicial.
- **aList.foldRight(initial_value)(binary_operator_function)**: Su esencia es la misma que la de *foldLeft*, pero recorriendo los elementos de la lista de derecha a izquierda. El primer parámetro de la función de operador binario es el elemento de la lista. El segundo, el acumulador.
- **aList.map(unary_operator_function)**: La función *map* aplicada sobre una lista, devuelve una nueva lista, resultado de aplicar una función sobre todos y cada uno de los elementos de la lista de izquierda a derecha.
- **aList.flatten**: El método *flatten* tiene como objetivo convertir una lista de listas en una única lista. Es preciso, por tanto, que *aList* sea una lista de listas.
- **aList.filter(predicate)**: La función *filter* devuelve una lista con los elementos de la lista original que cumplen con el predicado ($A \Rightarrow \text{Boolean}$). Si no hay ninguno, devuelve la lista vacía

ASIGNATURA: PROGRAMACIÓN DECLARATIVA	 <p>ACADEMIA UNIVERSITARIA</p> <p>Avenida Alcalde de Móstoles 33 (Posterior) 28922 Móstoles (Madrid)</p> <p>91 664 67 20 - 717 716 540 </p> <p>info@academia-atica.com www.academia-atica.com</p>
TITULACIÓN: INFORMÁTICA	
PROFESOR: JAVIER VERGARA IGUAL	
<h1>SCALA</h1>	

- **aList.size**: Devuelve el número de elementos de la lista. Es equivalente a *length*.
- **aList.sortWith((elem1, elem2) => order)**: Devuelve la lista original ordenada en función del *order* entre cada par de elementos.
- **aList.take(n)**: Devuelve los primeros n elementos de la lista original.
- **aList.distinct**: Elimina los repetidos de una lista.
- **aList.flatMap(unary_operator_function)**: La función *flatMap* es una combinación entre *map* y *flatten*. Primero aplica la función a cada elemento de la lista original y, por último, genera una única lista con el contenido de cada una de las sublistas. **Nota**: Si la lista original no es una lista de listas, debemos generar una sublista por cada elemento en la aplicación de la función *map*.

ATICA

ASIGNATURA: PROGRAMACIÓN DECLARATIVA


TITULACIÓN: INFORMÁTICA

PROFESOR: JAVIER VERGARA IGUAL

SCALA



Avenida Alcalde de Móstoles 33 (Posterior)
28922 Móstoles (Madrid)



91 664 67 20 - 717 716 540 

info@academia-atica.com www.academia-atica.com

ÁRBOLES

La función **foldTree** no existe en Scala, es decir, es “nuestra”. Se fundamenta en la estrategia de “divide y vencerás”. El parámetro *empty*, representa la solución directa a devolver cuando el árbol está vacío. El parámetro *node* especifica cuál es la forma de resolver el problema desde las sub-soluciones de los problemas correspondientes a los subárboles izquierdo y derecho.



```
def foldTree[A, B](tree: Tree[A])(empty: B)(node: (B, A, B) => B): B =  
  tree match {  
    case Empty() =>  
      empty  
    case Node(left, root, right) =>  
      node(foldTree(left)(empty)(node), root, foldTree(right)(empty)(node))  
  }
```

ASIGNATURA: PROGRAMACIÓN DECLARATIVA	 <p>ACADEMIA UNIVERSITARIA</p> <p>Avenida Alcalde de Móstoles 33 (Posterior) 28922 Móstoles (Madrid)</p> <p>91 664 67 20 - 717 716 540 </p> <p>info@academia-atika.com www.academia-atika.com</p>
TITULACIÓN: INFORMÁTICA	
PROFESOR: JAVIER VERGARA IGUAL	
<h1>SCALA</h1>	

MAPAS

Un mapa (**Map**) es una colección de datos que almacena parejas (Clave, Valor). Cada valor está identificado unívocamente por su clave. La consulta y el borrado se llevan a cabo indicando la clave de búsqueda.

- **aMap.values:** Devuelve un *Iterable* con todos los valores, excluyendo las claves del mapa. Para convertir el *Iterable* en una lista usamos la función *toList*.
- **aMap.keys:** Devuelve un *Iterable* con las claves. Para obtener una lista se aplica el método *toList*.
- **aMap.get(key):** Dada una clave, devuelve *None* si no existe ninguna correspondencia, o un *Option* en caso contrario. También podemos transformar el resultado en una lista usando *toList*.
- **aMap.filter(predicate):** Recorre cada pareja (clave, valor) del mapa y si se cumple el predicado, se queda con la pareja. Devuelve un Map con los elementos filtrados. El par es una dupla. Para acceder a cada una de sus partes usamos *._1* o *._2*.
- **aMap.map(unary_operator_function):** Devuelve un nuevo mapa aplicando la función a cada una de las parejas del original.

ASIGNATURA: PROGRAMACIÓN DECLARATIVA	 ACADEMIA UNIVERSITARIA Avenida Alcalde de Móstoles 33 (Posterior) 28922 Móstoles (Madrid) 91 664 67 20 - 717 716 540  info@academia-atica.com www.academia-atica.com
TITULACIÓN: INFORMÁTICA	
PROFESOR: JAVIER VERGARA IGUAL	
<h1>SCALA</h1>	

OTROS DETALLES

- El operador de comparación de igualdad es `==`.
- El tipo **Option[A]** consta de dos constructores **None** y **Some**. Por ejemplo, si indicamos, por ejemplo, que una función devuelve un valor del tipo *Option[Int]*, estamos expresando que puede devolver algo (*Some*), es decir, un valor de tipo entero, o nada (*None*).
- El tipo **Either[A, B]** nos permite escoger entre dos valores cuyos tipos pueden ser diferentes, es decir, nos permite devolver el uno o el otro. Tenemos el constructor **Left(A)** y el constructor **Right(B)**.

ATICA