UNIVERSIDAD DE MÁLAGA
Dpto. Lenguajes y Ciencias de la Computación

Programación avanzada II
Lab 3. Uso del API List

En la implementación de las funciones de la práctica, salvo que se indique lo contrario, deben realizarse utilizando las funciones de la clase `List` de Scala (`foldRight`, `foldLeft`, `map`,…). Las funciones `foldRight` y `foldLeft` se definen del modo siguiente:

```
def foldRight[B](acc:B)(f:(A,B)=>B):B = l match
  case Nil => acc
  case a::r => f(a,r.foldRight(acc)(f))

def foldLeft[B](acc:B)(f:(B,A)=>B):B = l match
  case Nil => acc
  case a::r => r.foldLeft(f(acc,a))(f)
```

1. Utilizando `foldRight`, define las funciones
   ```
   def sum(l:List[Int]):Int
   def product(l:List[Int]):Int
   def length[A](l:List[A]):Int
   ```
   que, respectivamente, suman/multiplican los elementos de la lista l, y calculan su longitud, respectivamente. Ejemplos:
   ```
   sum(List(1,2,3)) == 6
   product(List(1,3,5)) == 15
   length(List("Hola", " ", "Mundo")) == 3
   ```

2. Utilizando `foldLeft` o `foldRight` define las funciones
   ```
   def reverse[A](l:List[A]):List[A]
   def append[A](l1:List[A],l2:List[A]):List[A]
   ```
   que calculan la longitud de la lista l y la invierten. Ejemplos:
   ```
   reverse(List(1,2,3)) == List(3,2,1)
   append(List(1,2,3),List(1,2)) == List(1,2,3,1,2)
   ```

3. Utilizando `foldLeft` o `foldRight` define la función
   ```
   def existe[A](l:List[A],f:A=>Boolean):Boolean
   ```
   que comprueba si l tiene un elemento que satisface f. Ejemplos:
   ```
   existe(List(1,2,3),_>2) == true
   existe(List("Hola","Mundo"),_.length>=5) == true
   existe(List("Hola","Mundo"),_.length<3) == false
   ```

4. Define la función
   ```
   def f(l:List[Int]):List[Int]
   ```
   que dada la lista l construye una lista con los valores absolutos de los elementos negativos de l. Por ejemplo,
   ```
   f(List(1,-2,3,-4,-5,6)) == List(2,4,5)
   ```
   Implementa la función de dos formas
   a) Mediante una función recursiva de cola, haciendo uso del pattern matching
   b) Usando únicamente funciones de orden superior (map, filter, etc.)

5. Usando `foldRight` implementa la función
   ```
   def unzip[A](l:List[(A,B)]):(List[A],List[B])
   ```
   que dada una lista de tuplas `List((a1, b1), …, (an, bn))` devuelve dos listas de la forma `List(a1, …, an)` y `List(b1, …, bn)`. Ejemplo:
   ```
   unzip(List((1,'a'),(2,'b'),(3,'c'))) == (List(1,2,3), List('a','b','c'))
   ```

6. Usando `foldRight` implementa la función
   ```
   def compose[A](lf:List[A=>A],v:A):Boolean
   ```
   que dada una lista de funciones `List(f1,f2,..,fn)` y un valor `v` calcula `f1(f2(…fn(v))…)`. Por ejemplo,
   ```
   compose(List[Int => Int](Math.pow(_,2).toInt, _+2), 5) == (5+2)^2 == 49
   ```

7. Usando `foldRight` implementa la función
   ```
   def remdups[A](lista:List[A]):List[A]
   ```
   que elimina los duplicados adyacentes de la `lista`. Por ejemplo,
   ```
   remdups(List(1,1,3,3,3,2,1,2,2,1,2)) == List(1, 3, 2, 1, 2, 1, 2)
   ```

8. Usando `foldRight` implementa la función
   ```
   def fibonnaci(n:Int):Int
   ```
   que dado un número n calcula el n-ésimo número de Fibonacci. Por ejemplo,
   ```
   fibonacci(5) == 5
   fibonacci(10) == 55
   ```

9. Usando `foldRight` implementa la función
   ```
   def inits[A](l:List[A):List[List[A]]
   ```
   que construye una lista con todas las listas prefijos de `l`. Por ejemplo,
   ```
   inits(List(1,2,3)) == List(List(),List(1),List(1,2),List(1,2,3))
   inits(List(3)) == List(List(),List(3))
   inits(List()) == List(List())
   ```

10. Escribe una función
    ```
    def halfEven(l1:List[Int],l2:List[Int]):List[Int]
    ```
    que toma dos listas de enteros como entrada y suma sus elementos l1(i) y l2(i). Si la suma l1(i) + l2(i) es par se divide por dos. En otro caso, se elimina de la lista resultante. Por ejemplo,
    ```
    halfEven(List(1,2,3,4),List(3,2,4)) == List(2,2)
    ```
    Implementa la función de dos formas
    c) Mediante una función recursiva de cola, haciendo uso del pattern matching
    d) Usando únicamente funciones de orden superior (map, filter, etc.)

11. Dada una lista de cadenas de caracteres, cada una de las cuales comienza con "ERROR", "INFO" o "WARNING", queremos (1) contar el número de mensajes de cada tipo y (2) extraer los mensajes de error y guárdelos en una lista.
    Dados los siguientes datos:
    ```
    val logs = List(
      "ERROR: Null pointer exception",
      "INFO: User logged in",
      "ERROR: Out of memory",
      "WARNING: Disk space low",
      "INFO: File uploaded",
      "ERROR: Database connection failed"
    )
    ```
    Para `logs`, la salida de (1) debe ser
    ```
    HashMap(WARNING -> 1, ERROR -> 3, INFO -> 2)
    ```
    y para (2)
    ```
    List(ERROR: Null pointer exception, ERROR: Out of memory, ERROR: Database connection failed)
    ```

12. Dada una lista de transacciones de ventas representadas como (productName, quantitySold, pricePerUnit), queremos (1) calcular los ingresos totales y (2) obtener la lista de las transacciones de cuantías (quantitySold) superiores (o iguales) a 100 ordenadas por su cuantía.
    Dados los siguientes datos:
    ```
    val sales = List(
      ("Laptop", 2, 1000.0),
    ```

```
    ("Mouse", 10, 15.0),
    ("Keyboard", 5, 50.0),
    ("Monitor", 3, 200.0),
    ("USB Drive", 20, 5.0)
  )
```

la salida esperada para (1) es

```
3100.0
```

y para (2):

```
List((Laptop,2000.0), (Monitor,600.0), (Keyboard,250.0), (Mouse,150.0),
(USB Drive,100.0))
```

13. Dada una lista de oraciones, queremos extraer las **palabras únicas** (es decir, eliminar duplicados), convertirlas a minúsculas y eliminar las palabras no significativas (como "a", "el", "es", "de", etc.).
    Dados los siguientes datos:

```
val sentences = Set(
  "Scala is a functional language",
  "The power of functional programming is great",
  "Functional programming is elegant"
)
val stopWords = Set("a", "the", "is", "of")
```

    dando la salida como un conjunto, esta sería

```
HashSet(programming, language, scala, power, elegant, functional, great)
```

14. Dada una lista de palabras, queremos contar la **frecuencia de cada palabra**.
    Dados los siguientes datos

```
val words = List("scala", "is", "awesome", "scala", "functional", "scala",
"is", "great")
```

    la salida esperada es

```
HashMap(is -> 2, awesome -> 1, scala -> 3, functional -> 1, great -> 1)
```

15. Dados dos maps que representan el stock de productos en dos almacenes diferentes, queremos combinarlos **sumando las cantidades de** los productos que aparecen en ambos.
    Dados los siguientes datos

```
val warehouse1 = Map("laptop" -> 5, "mouse" -> 20, "keyboard" -> 10)
val warehouse2 = Map("laptop" -> 3, "mouse" -> 15, "monitor" -> 8)
```

    la salida esperada es

```
Map(laptop -> 8, mouse -> 35, keyboard -> 10, monitor -> 8)
```