Práctica 9: Programación funcional en Scala

Importante: en todos los ejercicios define explícitamente el tipo devuelto por las funciones, no dejes que sea el compilador de Scala el que lo infiera. De esta forma podrás comprobar si has cometido algún error de tipos en los valores devueltos por las funciones.

Ejercicio 1

a) Define en Scala la función recursiva pura (sin *tail recursion*) `aplica3D` que reciba una lista de coordenadas 3D en forma de tupla, una función unaria y una coordenada ("x", "y" ó "z"). Deberá aplicar la función a los elementos correspondientes a esa coordenada y devolver las nuevas coordenadas. Ejemplo:

```
def suma2(x: Int) = x + 2
assert(aplica3D(List((1,2,3), (4,5,6), (7,8,9), (10,11,12)), suma2 _, "y") ==
          List((1,4,3), (4,7,6), (7,10,9), (10,13,12)))
```

b) Define la función anterior utilizando funciones de orden superior.

Ejercicio 2

a) Escribe en Scala utilizando funciones de orden superior la función `intercambia(lista)` que reciba una lista de tuplas de dos elementos de tipo entero y devuelva una lista con las mismas tuplas pero con los elementos de cada tupla intercambiados.

Ejemplo:

```
assert(intercambia(List((1,2), (3,4), (5,6))) == List((2,1), (4,3), (6,5)))
```

b) Escribe en Scala la función `compruebaParejas(lista, func)` que reciba una lista de enteros y una función que recibe un entero y devuelve un entero, y devuelva una lista de tuplas. La función debe devolver en forma de tupla aquellos números contiguos que cumplan que el número de la derecha es el resultado de aplicar la función al número de su izquierda.

Ejemplo:

```
assert(compruebaParejas(List(2, 4, 16, 5, 10, 100, 105), (x)=>{x*x}) ==
       List((2,4), (4,16), (10,100)))
```

Ejercicio 3

a) Define utilizando un tipo genérico la función `minimo` que devuelve el valor mínimo de una lista.

Un ejemplo de su funcionamiento:

```
assert(minimo(List(100,-10,30,200), (x: Int, y: Int) => {x < y}) == -10)
assert(minimo(List("hola", "adios", "mola"), (x: String, y: String) => {x < y})
       == "adios")
```

b) Define utilizando tipos genéricos la función `minimaTupla` que recibe una lista de tuplas y una función `toInt` que convierte una tupla en entero y que devuelve la tupla con la que `toInt` devuelve el entero más pequeño.

Por ejemplo, podemos aplicar `minimaTupla` a una lista de tuplas `(String, Int)` pasando una función `toInt` que suma la longitud de la cadena al entero de la tupla:

```
assert(minimaTupla(List(("hola",10),("pep",9),("supercalifragilistcoespialidoso",2)
                  (t: (String, Int)) => {t._1.length + t._2}) == ("pep",9))
```

Ejercicio 4

Implementa utilizando funciones de orden superior la función

```
masFrecuentes(s: String, n: Int): List[String]
```

que recibe una frase con palabras separadas por espacios y devuelve una lista con las *n* cadenas que más veces aparecen en la frase, ordenadas de mayor número de apariciones a menor. En el caso en que las palabras aparezcan el mismo número de veces, aparecerán por orden alfabético. Las palabras mayúsculas y minúsculas se considerarán iguales. Puedes definir funciones auxiliares y cualquier método de la clase List (enlace).

Ejemplo:

```
assert(masFrecuentes("En esta frase hay Mas de mas Frase en repetidas de mas De", 4
       List("de", "mas", "en", "frase"))
```

Pista 1: Puedes transformar la cadena en un cadena con todas las palabras en minúscula.
Pista 2: Para calcular las *n* palabras más frecuentes será necesario calcular una lista de tuplas `(String, Int)` que guarde la frecuencia de cada cadena.

Ejercicio 5

a) Implementa la función `creaOperador` que recibe un entero (*valor inicial*) y una función de dos enteros que devuelve otro entero (*operacion*) y devuelve una clausura de un argumento en la que se aplica *operacion* a *valor inicial* y el argumento.

Ejemplo:

```
assert(creaOperador(10, _ + _)(100) == 110)
assert(creaOperador(10, (x,y) => {y/x})(200) == 20)
```

b) Implementa utilizando tipos genéricos la función `creaIterador` que recibe una frase (palabras separadas por espacios), un caso base y una función (*procesaPalabra*) y devuelve una clausura con estado local (*iterador*). Cada invocación a *iterador* aplica la función *procesaPalabra* a una palabra de la cadena, devuelve el resultado y modifica el estado local para que la cadena avance y la próxima palabra sea la siguiente de la cadena. Cuando se ha procesado todo la cadena se devuelve el caso base.

```
val it1 = creaIterador("En un lugar", '%', _.head)
assert(it1() == 'E')
assert(it1() == 'u')
assert(it1() == 'l')
assert(it1() == '%')
assert(it1() == '%')

val it2 = creaIterador("En un lugar", -1, _.length)
assert(it2() == 2)
assert(it2() == 2)
assert(it2() == 5)
assert(it2() == -1)
assert(it2() == -1)
```
