

---

# Práctica 2: programación funcional en Scala; recursividad

## Nuevas tecnologías de la programación

---

### Contenido:

1	Objetivos	1
2	Primera parte: funciones	1
2.1	Triángulo de Pascal . . . . .	1
2.2	Balanceo de cadenas con paréntesis . . . . .	3
2.3	Contador de posibles cambios de moneda . . . . .	4
2.4	Busqueda binaria genérica . . . . .	5
3	Funciones sobre clase lista propia	5
4	Implementación	8
5	Material a entregar	9

---

## 1 Objetivos

En esta segunda práctica se trata de trabajar con el lenguaje de programación Scala, definiendo algunas funciones recursivas cuya declaración ya se proporciona. También introducir el uso de algunos elementos del lenguaje como objetos, interfaces, clases tipo **case** y la librería de pruebas **scalacheck**. La implementación realizada debe ser capaz de superar un conjunto de pruebas desarrollado por vosotros pero que garantice el correcto funcionamiento del código.

## 2 Primera parte: funciones

### 2.1 Triángulo de Pascal

El siguiente patrón de números se conoce como triángulo de Pascal:

```
      1
     1 1
    1 2 1
   1 3 3 1
  1 4 6 4 1
 1 5 10 10 5 1
```

		1	6	15	20	15	6	1		
	1		7	21	35	35	21	7	1	
	1	8	28	56	70	56	28	8	1	
1	9	36	84	126	126	84	36	9	1	
1	10	45	120	210	252	210	120	45	10	1

Los números en el vértice del triángulo son todos 1 y cada número interior puede obtenerse como la suma de los valores que tiene sobre él. Se trata de escribir una función que calcule los elementos del triángulo de forma recursiva. Para ello se escribirá una función con la siguiente declaración:

```
1 def calcularValorTrianguloPascal(columna: Int, fila: Int): Int
```

Esta función recibe como argumento una columna y una fila (comenzando por el valor 0) y devuelve el valor almacenado en la posición correspondiente del triángulo. Esto permitiría escribir un método **main** de la forma siguiente:

```
1  /**
2   * Metodo main: en realidad no es necesario porque el desarrollo
3   * deberia guiarse por los tests de prueba
4   *
5   * @param args
6   */
7  def main(args: Array[String]) {
8      println("..... Triangulo de Pascal .....")
9
10     // Se muestran 10 filas del trinagulo de Pascal
11     for (row <- 0 to 10) {
12         // Se muestran 10 y 10 filas
13         for (col <- 0 to row)
14             print(calcularValorTrianguloPascal(col, row) + " ")
15
16         // Salto de linea final para mejorar la presentacion
17         println()
18     }
19
20     // Se muestra el valor que debe ocupar la columna 5 en la fila 10
21     print(calcularValorTrianguloPascal(10, 15))
22 }
```

que permite obtener la siguiente salida:

```
..... Triangulo de Pascal .....
1
1 1
1 2 1
1 3 3 1
```

```

1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
3003

```

Aunque se use el **main**, como se ha comentado antes, la forma básica de probar consistirá en usar el conjunto de casos de prueba específico para esta función:

- los valores en los laterales del triángulo son siempre uno: aquellos que ocupan la primera y última posición de cada fila
- los valores internos siempre deben ser iguales a la suma de los dos elementos superiores

Como ejemplo, se incluye la forma de comprobar la primera de las propiedades:

```

1 // Se generan los valores de fila y columna para los bordes
2 val coordenadasExtremos = for {
3     fila <- Gen.choose(0, MAXIMO)
4     columna <- Gen.oneOf(0, fila)
5 } yield (fila, columna)
6
7 property("Elementos en lados del triangulo valen 1") = {
8     forAll(coordenadasExtremos) { (i) => {
9         val resultado=PracticaFunciones.calcularValorTrianguloPascal(i._1, i._2)
10        println("Fila = "+i._1+" Columna = "+ i._2+ " Resultado = "+resultado)
11        resultado == 1
12    }
13 }
14 }

```

## 2.2 Balanceo de cadenas con paréntesis

Se trata aquí de escribir una función recursiva que verifique el balanceo de los paréntesis presentes en una cadena de caracteres, representada como **List[Char]** (no como objeto de la clase **String**). Algunas cadenas balanceadas son:

- (if (a ¿ b) (b/a) else (a/(b\*b)))
- (ccc(ccc)cc((ccc(c))))

Algunas no balanceadas:

- (if (a ¿ b) b/a) else (a/(b\*b)))

- `(ccc(ccccc((ccc(c)))))`
- `()()()`
- `()(`

El último ejemplo pone de manifiesto que no es suficiente verificar que la expresión contiene el mismo número de paréntesis abriendo y cerrando, ya que deben seguir el orden adecuado. La función tendrá la siguiente declaración:

```
1 def chequearBalance(cadena: List[Char]): Boolean
```

Hay tres métodos de la clase **List** que son útiles para realizar este ejercicio:

- **cadena.isEmpty**: comprueba si la lista está vacía
- **cadena.head**: obtiene el primer elemento de la lista
- **cadena.tail**: devuelve una nueva lista sin el primer elemento

Pueden definirse funciones auxiliares, si resulta conveniente. Para que una cadena esté bien formada debe cumplirse que en cualquier subcadena de la cadena a probar, tomada desde el principio, el número de caracteres '(' menos el número de caracteres ')' nunca debe ser negativo.

La prueba de este método puede basarse en la generación de cadenas aleatorias que contengan paréntesis, que podrían generarse mediante el siguiente código:

```
1 // Generacion de cadenas de longitud n: forma de uso strGen(10) para cadenas
2 // de 10 caracteres
3 val strGen =
4   (n: Int) =>
5     Gen.listOfN(n, Gen.oneOf('(', ')', Gen.alphaChar.sample.get)).
6     map(_.mkString)
```

## 2.3 Contador de posibles cambios de moneda

Se trata aquí de escribir una función recursiva que determine de cuántas formas posibles puede devolverse una cierta cantidad. Por ejemplo, con monedas de valor 1 y 2 hay 3 formas de cambiar el valor 4:

- $1 + 1 + 1 + 1$
- $1 + 1 + 2$
- $2 + 2$

La función tiene la siguiente declaración:

```
1 def contarCambiosPosibles(cantidad: Int, monedas: List[Int]): Int
```

Aquí pueden usarse también los métodos de utilidad vistos en el ejercicio anterior y relativos a la clase **List**. En este ejercicio ayuda pensar en los casos extremos:

- ¿cuántas formas hay de dar cambio de un valor 0?
- ¿cuántas formas hay de dar cambio de un valor positivo si no tenemos monedas?

La lista que recibe la función como argumento contiene el tipo de monedas que pueden usarse para el cambio (pensad en tipos de monedas y no en número de monedas). Los cambios en la lista se hacen atendiendo a la posibilidad de usar los tipos de monedas para seguir dando cambio. Por ejemplo, imaginad que en un determinado instante del proceso de solución se llega a la cantidad 2 a devolver. A la hora de plantear cómo seguir procesando el cambio de esta cantidad ya no tiene sentido considerar las monedas de valor 3, por lo que no se usarían a partir de dicho instante.

Debéis pensar en la especificación de las pruebas para este problema basada en la especificación de propiedades. Una posibilidad podría ser modificar el método para que devolviese una lista con todos los posibles cambios (cada elemento de la lista sería una forma de cambio) y comprobar que todas ellas suman la cantidad a devolver.

## 2.4 Búsqueda binaria genérica

En este ejercicio se debe implementar un método genérico (parametrizado) de búsqueda binaria. El método debe ser recursivo gracias al uso de una función auxiliar interna que soporte la anotación propia de la recursión por la cola. La propiedad a cumplir por este método es sencilla: dada cualquier lista aleatoria de valores de tipo entero, por ejemplo, el resultado producido por el método empleado y el método de búsqueda propio de la clase lista deben coincidir.

```
1 def busquedaBinaria[A](coleccion : Array[A], aBuscar: A,
2                       criterio : (A,A) => Boolean) : Int = ???
```

## 3 Funciones sobre clase lista propia

Esta segunda parte de la práctica consistirá en jugar con la declaración de una clase propia de gestión de listas declarada usando los siguientes elementos:

```
1 /**
2  * Interfaz generica para la lista
3  * @tparam A
4  */
5 sealed trait Lista[+A]
6
```

```

7  /**
8   * Objeto para definir lista vacia
9   */
10 case object Nil extends Lista[Nothing]
11
12 /**
13  * Clase para definir la lista como compuesta por elemento inicial
14  * (cabeza) y resto (cola)
15  * @param cabeza
16  * @param cola
17  * @tparam A
18  */
19 case class Cons[+A](cabeza : A, cola : Lista[A]) extends Lista[A]

```

A partir de estos elementos y en el cuerpo de un objeto denominado **Lista** se trata de implementar los siguientes métodos (algunos de ellos fueron implementados en clase):

```

1  /**
2   * Metodo para permitir crear listas sin usar new (hecho en clase)
3   * @param elementos secuencia de elementos a incluir en la lista
4   * @tparam A
5   * @return
6   */
7  def apply[A](elementos : A*) : Lista[A] = ???
8
9  /**
10   * Obtiene la longitud de una lista
11   * @param lista
12   * @tparam A
13   * @return
14   */
15  def longitud[A](lista : Lista[A]) : Int = ???
16
17  /**
18   * Metodo para sumar los valores de una lista de enteros
19   * @param enteros
20   * @return
21   */
22  def sumaEnteros(enteros : Lista[Int]) : Double = ???
23
24  /**
25   * Metodo para multiplicar los valores de una lista de enteros
26   * @param enteros
27   * @return
28   */
29  def productoEnteros(enteros : Lista[Int]) : Double = ???
30
31  /**
32   * Metodo para agregar el contenido de dos listas
33   * @param lista1
34   * @param lista2

```

```

35  * @tparam A
36  * @return
37  */
38  def concatenar[A](lista1: Lista[A], lista2: Lista[A]): Lista[A] = ???
39
40  /**
41   * Funcion de utilidad para aplicar una funcion de forma sucesiva a los
42   * elementos de la lista con asociatividad por la derecha
43   * @param lista
44   * @param neutro
45   * @param funcion
46   * @tparam A
47   * @tparam B
48   * @return
49   */
50  def foldRight[A, B](lista : Lista[A], neutro : B)(funcion : (A, B) => B): B = ???
51
52  /**
53   * Suma mediante foldRight
54   * @param listaEnteros
55   * @return
56   */
57  def sumaFoldRight(listaEnteros : Lista[Int]) : Double = ???
58
59  /**
60   * Producto mediante foldRight
61   * @param listaEnteros
62   * @return
63   */
64  def productoFoldRight(listaEnteros : Lista[Int]) : Double = ???
65
66  /**
67   * Reemplaza la cabeza por nuevo valor. Se asume que si la lista esta vacia
68   * se devuelve una lista con el nuevo elemento
69   *
70   * @param lista
71   * @param cabezaNueva
72   * @tparam A
73   * @return
74   */
75  def asignarCabeza[A](lista : Lista[A], cabezaNueva : A) : Lista[A] = ???
76
77  /**
78   * Elimina el elemento cabeza de la lista
79   * @param lista
80   * @tparam A
81   * @return
82   */
83  def tail[A](lista : Lista[A]): Lista[A] = ???
84
85  /**
86   * Elimina los n primeros elementos de una lista
87   * @param lista lista con la que trabajar
88   * @param n numero de elementos a eliminar

```

```

89  * @tparam A tipo de datos
90  * @return
91  */
92  def eliminar[A](lista : Lista[A], n: Int) : Lista[A] = ???
93
94  /**
95   * Elimina elementos mientras se cumple la condicion pasada como
96   * argumento
97   * @param lista lista con la que trabajar
98   * @param criterio predicado a considerar para continuar con el borrado
99   * @tparam A tipo de datos a usar
100  * @return
101  */
102  def eliminarMientras[A](lista : Lista[A], criterio: A => Boolean) : Lista[A] = ???
103
104
105  /**
106   * Elimina el ultimo elemento de la lista. Aqui no se pueden compartir
107   * datos en los objetos y hay que generar una nueva lista copiando
108   * datos
109   * @param lista lista con la que trabajar
110   * @tparam A tipo de datos de la lista
111   * @return
112  */
113  def eliminarUltimo[A](lista : Lista[A]) : Lista[A] = ???
114
115  /**
116   * foldLeft con recursividad tipo tail
117   * @param lista lista con la que trabajar
118   * @param neutro elemento neutro
119   * @param funcion funcion a aplicar
120   * @tparam A parametros de tipo de elementos de la lista
121   * @tparam B parametro de tipo del elemento neutro
122   * @return
123  */
124  @annotation.tailrec
125  def foldLeft[A, B](lista : Lista[A], neutro: B)(funcion : (B, A) => B): B = ???

```

Los casos de prueba de esta parte de la práctica deben basarse en la definición de propiedades que comprueben que los métodos implementados funcionan de la misma forma que los aportados por la clase **List**.

## 4 Implementación

Todo el código puede desarrollarse usando como punto de partida el esqueleto disponible en el material de la práctica, con nombre **Funciones.Scala** y **Lista.scala**, rellenando el cuerpo de las funciones con las sentencias necesarias. Recordad que es posible incluir funciones dentro de funciones (para el caso en que convenga usar alguna función auxiliar para resolver el problema). Podéis utilizar las funciones auxiliares que consideréis oportunas para facilitar la implementación de la funcionalidad o pruebas pedidas.



## 5 Material a entregar

Al final de la realización de la práctica se entregará un archivo comprimido con el contenido completo de la práctica, tal y como se integra en el proyecto con el entorno de desarrollo que hayáis usado. Se incluirá también un pequeño documento indicando el entorno de desarrollo y una breve valoración de la práctica (si los conceptos vistos son novedosos, si os ha parecido de interés, problemas encontrados, etc) en tres o cuatro líneas.

La fecha de entrega se fijará en unos días. La entrega se hará mediante la plataforma **PRADO**.