# Práctica 2: programación funcional en Scala - recursividad

Nuevas tecnologías de la programación

## Contenido:



# 1 Objetivos

En esta segunda práctica se trata de trabajar con el lenguaje de programación **Scala**, definiendo algunas funciones recursivas. También se pretende introducir el uso de algunos librerías de pruebas como **Scalatest y Scalacheck** y de medida de tiempos de ejecución (como **Scalameter**). Usando estas herramientas, la implementación a realizar debe ser capaz de superar un conjunto de pruebas desarrollado por vosotros pero que garantice el correcto funcionamiento del código.

Asumiendo que se usa **IntelliJ**, para integrar estas librerías en el proyecto se recomienda crear el proyecto de tipo **sbt** y hacer que el contenido del archivo **build.sbt** sea similar al siguiente (el nombre del proyecto es libre; ajustad las versiones en caso de ser necesario):

```
name := "practica2"

version := "0.1"

scalaVersion := "2.13.5"

libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.14.1" % "test"

libraryDependencies += "org.scalactic" %% "scalactic" % "3.2.7"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.2.7" % "test"
resolvers += "Sonatype OSS Snapshots" at "https://oss.sonatype.org/content/repositories/
libraryDependencies += "com.storm-enroute" %% "scalameter" % "0.21"

testFrameworks += new TestFramework("org.scalameter.ScalaMeterFramework")

parallelExecution in Test := false
```

Se describen a continuación las funciones recursivas a implementar.

# 2 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 los lados 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(fila: Int, columna: Int): Int
```

Esta función recibe como argumento una fila y una columna (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:

```scala
  /**
    * Metodo main: en realidad no es necesario porque el desarrollo
    * deberia guiarse por los tests de prueba
    *
    * @param args
    */
  def main(args: Array[String]) {
    println("..................... Triangulo de Pascal ...................")

    // Se muestran 10 filas del trinagulo de Pascal
    for (row <- 0 to 10) {
      // Se muestran 10 y 10 filas
      for (col <- 0 to row)
        print(calcularValorTrianguloPascal(row, col) + " ")

      // Salto de linea final para mejorar la presentacion
      println()
    }

    // Se muestra el valor que debe ocupar la fila 10 y columna 5
    print(calcularValorTrianguloPascal(10, 5))
  }
```

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
252
```

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. Las propiedades que cumplen los valores del triángulo son:

- 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 usando la librería **Scalacheck** (especialmente basada en la idea de comprobación de propiedades):

```
// Se generan los valores de fila y columna para los bordes
val coordenadasExtremos = for {
   // se genera numero de fila: valor comprendido entre 0 y MAXIMO
   // (MAXIMO no esta incluido)
   fila <- Gen.choose(0, MAXIMO)

   // se genera numero de columna: o 0 o el valor de fila. Esto
   // asegura que se trata de un valor de los extremos (X,0) o
   // (X,X)
   columna <- Gen.oneOf(0, fila)
} yield (fila, columna)

property("Elementos en lados del triangulo valen 1") = {
   forAll(coordenadasExtremos) { (i) => {
        val resultado = calcularValorTrianguloPascal(i._1, i._2)
        println("Fila = "+i._1 +" Columna = "+ i._2+ " Resultado = "+resultado)
        resultado == 1
      }
   }
}
```

Otro enfoque de prueba consiste en usar una librería típica de pruebas unitarias, como **Scalatest**. En este caso, el fragmento de código a usar podría ser similar al siguiente:

```
   // se asigna la funcion a usar, para el caso en que haya varias variantes
   val funcion = TrianguloPascal.calcularValorTrianguloPascalV1(_, _)

   // Prueba 1: se calcula el valor de la columna 0 y fila 2
   test("trianguloPascal: fila=2, columna=0") {
      assert(funcion(2,0) === 1)
   }

   // Prueba 2: calculo del valor de columna 1 y fila 2
   test("trianguloPascal: fila=2, columna=1") {
      assert(funcion(2,1) === 2)
   }

   // Prueba 3: calculo de valor de columna 1, fila 3
   test("trianguloPascal: fila=3, columna=1") {
      assert(funcion(3,1) === 3)
   }

   // Prueba 4: calculo de valor de columna 5, fila 10
   test("trianguloPascal: fila=10, columna=5") {
      assert(funcion(10,5) === 252)
   }

   // Prueba 5: calculo del valor de columna 10 y fila 15
```

```
25    test("trianguloPascal: fila=15, columna=10") {
26       assert(funcion(15, 10) === 3003)
27    }
28
29    // Prueba 5: calculo del valor de columna 0 y fila 0
30    test("trianguloPascal: fila=0, columna=0") {
31       assert(funcion(0,0) === 1)
32    }
```

Una forma eficiente de implementar la función puede basarse en la relación siguiente: el número que ocupa la fila $f$ y columna $c$ puede calcularse mediante la siguiente ecuación:

$$\binom{f}{c} = \frac{f!}{c!(f-c)!} \tag{1}$$

Por ejemplo, comprobamos que el valor de la fila 6 y columna 2, cuyo valor es 15, se obtiene como:

$$\binom{6}{2} = \frac{6!}{2!(4)!} = 15$$

# 3 Series definidas de forma recurrente

Hay muchas series de números definidas de forma recurrente, donde el término que ocupa una determinada posición viene dado por alguna combinación de números previos. Se indican a continuación algunos ejemplos:

- **Fibonacci**: los dos primeros términos son 0 y 1 y partir de ahí el término $i-esimo$ se define como $l_i = l_{i-1} + l_{i-2}$
- **Lucas**: los dos primeros términos son ahora 2 y 1 y $l_i = l_{i-1} + l_{i-2}$
- **Pell**: $l_0 = 2$, $l_1 = 6$, $l_i = 2 * l_{i-1} + l_{i-2}$
- **Pell-Lucas**: $l_0 = 2$, $l_1 = 2$, $l_i = 2 * l_{i-1} + l_{i-2}$
- **Jacobsthal**: $l_0 = 0$, $l_1 = 1$, $l_i = l_{i-1} + 2 * l_{i-2}$

Se trata, por tanto, de definir una función genérica que permita generar todas estas secuencias o cualquier otra que pueda definirse mediante dos valores iniciales y una relación que permita calcular el valor siguiente a partir de dos valores previos. Si es posible la función debería definirse de forma que la recursividad pudiera realizarse de forma efectiva (*tail recursion*).

Las pruebas asociadas a esta sección quedan a vuestra elección (no solo las pruebas, sino también el enfoque basado en propiedades o en comprobaciones directas).

# 4 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 balancedas 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 implementación de esta función se basará en devolver dos contadores: uno para '(' y otro para ')'.

La prueba de este método puede basarse en la generación de cadenas aleatorias que contengan paréntesis, que podrían generarse mediante **Scalacheck** con 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)
```

```
7
8  property("Balance de cadenas") = {
9        forAll(strGen(Gen.choose(1,MAXIMALONGITUD).sample.get)) {
10          cadena => {
11             val condicion = chequearBalance(cadena.toList)
12             var global = true
13             for(i <- 2 until cadena.length) {
14                val substring=cadena.substring(0,i)
15                val openCount=substring.filter(c => c == '(').length
16                val closeCount=substring.filter(c => c == ')').length
17                global = global && ((openCount-closeCount) >= 0)
18             }
19
20             // si se cumple la condicion, entonces global debe
21             // ser true
22             if (condicion == true) global == true
23             // en caso de no cumplirse, la condicion global puede
24             // ser positiva o negativa....
25             else true
26          }
27       }
28    }
```

La forma de probar esta funcionalidad mediante **Scalatest** podría ser:

```
1  import org.scalatest.funsuite.AnyFunSuite
2
3  import ContadorParentesis.chequearBalance
4
5  class ContadorParentesisTest extends AnyFunSuite {
6
7     // Prueba 1
8     test("chequear balance: '(if (zero? x) max (/ 1 x))' esta balanceada") {
9        assert(chequearBalance("(if (zero? x) max (/ 1 x))".toList))
10    }
11
12    // Prueba 2
13    test("chequear balance: 'Te lo dije ...' esta balanceada") {
14       assert(chequearBalance("Te lo dije (eso esta (todavia) hecho))".toList))
15    }
16
17    // Prueba 3
18    test("chequear balance: ':-)' no esta balanceada") {
19       assert(!chequearBalance(":-)".toList))
20    }
21
22    // Prueba 4
23    test("chequear balance: no basta con contar sin mas") {
24       assert(!chequearBalance("())(".toList))
25    }
26
27    // Prueba 5
28    test("(if (a > b) (b/a) else (a/(b*b)))"){
```

```
29          assert(chequearBalance("(if (a > b) (b/a) else (a/(b*b)))".toList))
30      }
31
32      // Prueba 6
33      test("(ccc(ccc)cc((ccc(c))))"){
34          assert(chequearBalance("(ccc(ccc)cc((ccc(c))))".toList))
35      }
36
37      // Prueba 7
38      test("(if (a > b) b/a) else (a/(b*b)))"){
39          assert(!chequearBalance("(if (a > b) b/a) else (a/(b*b)))".toList))
40      }
41
42      // Prueba 7
43      test("(ccc(ccccc((ccc(c))))"){
44          assert(!chequearBalance("(ccc(ccccc((ccc(c))))".toList))
45      }
46
47      // Prueba 8
48      test("())()())"){
49          assert(!chequearBalance("())()())".toList))
50      }
51  }
```

# 5 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 listarCambiosPosibles(cantidad: Int,
2                            monedas: List[Int]): List[List[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.

El método devuelve una lista con todos los posibles cambios (cada elemento de la lista sería una forma de cambio) y las pruebas pueden basarse en comprobar que todas ellas suman la cantidad a devolver.

# 6 Búsqueda genérica en colecciones ordenadas

## 6.1 Búsqueda binaria

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 = ???
```

## 6.2 Búsqueda a saltos genérica

Se trata de implementar un método genérico de búsqueda utilizando el algoritmo de búsqueda a saltos. Supongamos que se pretende buscar un valor en la colección

$$\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610\}$$

con 16 elementos. Supongamos también que el valor buscado es 55. La búsqueda se hará siguiendo los pasos:

- se asume un tamaño de bloque dado por la raíz cuadrada de la longitud de la colección: 4 en este caso
- se compara el elemento buscado con el final del primer bloque, elemento en posición 3. Como es mayor, saltamos el bloque
- se compara el elemento buscado con el final del segundo bloque, con último valor en índice 7. Como el valor buscado es mayor, entonces se salta de bloque
- se compara el elemento de referencia con el elemento final del tercer bloque, índice 11. Como 89 es mayor que el valor buscado, se realiza búsqueda lineal en el bloque actual (de 8 a 11).

# 7 Uso de paquetes de prueba y de medición de tiempos de ejecución

Las implementaciones de la práctica (hasta los métodos de búsqueda) deben probarse con la librería de pruebas que consideréis más oportuna. En clase y en el guión se han visto ejemplos de uso de **Scalatest** y **Scalacheck**. Como se indicó se trata de enfoques diferentes e igualmente útiles. Pensad en la forma más adecuada de ejecutar pruebas que garanticen el funcionamiento del código implementado.

Para los métodos de búsqueda debe implementarse también un conjunto de pruebas de ejecución que nos indique el tiempo aproximado de ejecución de cada una de las búsquedas, realizadas, por supuesto, sobre la misma colección de valores. Conviene probar con varios tamaños de colecciones y varios tipos de datos para extraer conclusiones sobre las características de cada algoritmo. Para esto podéis utilizar **Scalameter**.

# 8 Entrega de la práctica

## 8.1 Valoración

De los 6 problemas deben realizarse al menos 4, lo que permite optar hasta una nota máxima de 7. La realización de los 6 problemas permite optar a la nota máxima.

## 8.2 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**.