Práctica 3: programación funcional en Scala; clases y objetos

Nuevas tecnologías de la programación

Contenido:

1 Objetivos

En esta práctica se trabajará con una representación funcional de conjuntos basada en la noción matemática de funciones características. El objetivo de la práctica es poner en juego algunos de los conceptos de clases y objetos vistos en este tema.

2 Representación

En la práctica se trabaja, sin pérdida de generalidad, con conjuntos definidos por propiedades aplicadas sobre enteros. Como ejemplo motivador, pensemos en la forma de representar el conjunto de todos los enteros negativos: $x < 0$ sería la función característica. A la hora de definir este conjunto no tiene sentido recurrir a la enumeración de todos sus elementos, por razones obvias. Sin embargo, sí que es posible definir una característica que cumplirán todos ellos. En el caso de los números negativos la función característica, escrita en **Scala** sería:

```
1 (x : Int) => x < 0
```

Siguiendo esta idea, la representación del conjunto se hará definiendo la clase **Conjunto** que define un dato miembro que permite almacenar la función característica (función que recibe argumento un valor entero y devuelve un valor booleano indicando pertenencia o no):

```scala
/**
  * Clase para representar conjuntos definidos mediante una funcion
  * caracteristica (un predicado). De esta forma, se declara el tipo
  * conjunto como un predicado que recibe un entero (elemento) como
  * argumento y dvuelve un valor booleano que indica si pertenece o no
  * al conjunto
  *
  * @param funcionCaracteristica
  */
class Conjunto(val funcionCaracteristica: Int => Boolean) {
/**
    * Crea una cadena con el contenido completo del conjunto
    *
    * @return
    */
  override def toString(): String = {
    // El uso de this(i) implica la el uso del metodo apply
    val elementos = for (i <- -Conjunto.LIMITE to Conjunto.LIMITE
                         if this(i)) yield i
    elementos.mkString("{", ",", "}")
  }

  /**
    * Metodo para determinar la pertenencia de un elemento al
    * conjunto
    * @param elemento
    * @return valor booleano indicando si elemento cumple
    *         la funcion caracteristica o no
    */
  def apply(elemento: Int): Boolean = ???
}
```

Se aprecia que la clase dispone del método **apply**. Este método permite facilitar la comprobación de si un valor pertenece o no al conjunto representado por un objeto de la clase **Conjunto**:

```scala
// Creamos conjunto para valores mayores que 3
val conjunto=new Conjunto((x:Int) => x > 3)

// Se comprueba si 5 pertenece al conjunto
val pertenece=conjunto(5)
```

Se usa un objeto asociado (*companion object* a la clase, como forma de almacenar en él operaciones frecuentes sobre conjuntos. Entre otras cosas, el objeto aporta un dato miembro privado y constante llamado **LIMITE** cuyo papel se explica más adelante. La cabecera de la declaración del objeto es la siguiente:

```scala
/**
  * Objeto companion que ofrece metodos para trabajar con
  * conjuntos
  */
object Conjunto {
   /**
     * Limite para la iteracion necesaria algunas operaciones,
     * entre -1000 y 1000
     */
  private final val LIMITE = 1000

  /**
     * Metodo que permite crear objetos de la clase Conjunto
     * de forma sencilla
     * @param f
     * @return
     */
  def apply(f: Int => Boolean): Conjunto = {
    new Conjunto(f)
  }
.............................................
}
```

El dato miembro privado **LIMITE** se usará posteriormente en algunas operaciones que precisan iterar sobre los elementos del conjunto. En estas operaciones necesitamos limitar el número de valores sobre los que hacer las comprobaciones pertinentes. De modo práctico, podemos considerar que los conjuntos que estamos diseñando trabajarán únicamente con valores enteros en el intervalo $(-Limite, Limite)$. Puede apreciarse un ejemplo de aplicación en el método **toString** de la clase **Conjunto**.

El método **apply** asociado al objeto permite que la creación de objetos de la clase compañera pueda simplificarse, sin necesidad de usar **new**:

```scala
// Creamos conjunto para valores mayores que 3
// usando la facilidad del metodo apply del objeto
// compañero
val conjunto=Conjunto((x:Int) => x > 3)
```

3 Funciones básicas sobre conjuntos

Interesa dotar a esta definición de conjuntos de algunas operaciones básicas, como las siguientes (todas ellas se implementan el objeto y no en la clase):

- creación de conjunto con un único elemento. Este conjunto representa a este único elemento y su declaración será

  ```scala
  def conjuntoUnElemento(elemento : Int) : Conjunto = ???
  ```

- unión de dos conjuntos:

  ```scala
  def union(c1 : Conjunto, c2 : Conjunto) : Conjunto = ???
  ```

- intersección:

  ```scala
  def interseccion(c1 : Conjunto, c2 : Conjunto) : Conjunto = ???
  ```

- diferencia:

```
def diferencia(c1 : Conjunto, c2 : Conjunto) : Conjunto = ???
```

- filtrado:

```
def filtrar(c : Conjunto, predicado : Int => Boolean) : Conjunto = ???
```

4 Funciones avanzadas sobre conjuntos

Ahora estamos interesados en funciones que permitan hacer consultas sobre todos los elementos del conjunto. Como no hay forma directa de obtener todos los elementos de un conjunto, habrá que iterar sobre un determinado rango de enteros (dado por el límite definido en la clase **Conjunto**; para las pruebas conviene restringirlo y ampliarlo sólo para la versión final) para comprobar si pertenecen o no al conjunto de interés. Las funciones que deseamos implementar en esta sección son:

- paraTodo, que comprueba si un determinado predicado se cumple para todos los elementos del conjunto. La declaración es:

```
def paraTodo(c : Conjunto, predicado : Int => Boolean) : Boolean = ???
```

  Esta función debe implementarse de forma recursiva. Para ayudar a su desarrollo se ofrece un esqueleto de la misma:

```
def paraTodo(conjunto : Conjunto, predicado : Int => Boolean) : Boolean = {
  def iterar(elemento : Int) : Boolean = {
     if(???) ???
     else if (???) ???
     else predicado(elemento) && iterar(???)
  }
  iterar(-LIMITE)
}
```

- usando la función anterior, implementad la función existe que determine si un conjunto contiene al menos un elemento para el que se cumple un cierto predicado:

```
def existe(c : Conjunto, predicado : Int => Boolean) : Boolean = ???
```

- implementad una función **map** que transforme un conjunto en otro aplicando una cierta función:

```
def map(c : Conjunto, funcion : Int => Int) : Conjunto = ???
```

5 Observaciones

La mayor parte de las soluciones se pueden escribir como una línea única. En caso de no ser así seguramente debes repensar la solución. En definitiva, es una práctica en la que hay más que pensar (y dibujar en papel) que implementar.

Al igual que en prácticas anteriores el código implementado debe superar un determinado conjunto de test de prueba que garanticen su correcto funcionamiento. En PRADO disponéis de un archivo de ejemplo realizado para **scalatest** y otro con un ejemplo de pruebas usando **scalacheck**. Lo ideal sería disponer de conjuntis de pruebas en ambas plataformas. Se incluye también a continuación el contenido del archivo **build.sbt** para definir las dependencias necesarias:

```
libraryDependencies += "junit" % "junit" % "4.12"

libraryDependencies += "org.scalactic" %% "scalactic" % "3.0.5"
libraryDependencies += "org.scalatest" %% "scalatest" % "3.0.5" % "test"

libraryDependencies += "org.scalacheck" %% "scalacheck" % "1.13.4"% "test"
```

6 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 próximamente. La entrega se hará mediante la plataforma **PRADO**.