Práctica 3: programación funcional en Scala; clases y objetos

Nuevas tecnologías de la programación

Contenido:

1 Objetivos

En esta práctica se trabajará, de forma opcional., con diferentes estructuras de almacenamiento de información, con el objetivo de practicar los conceptos básicos de diseño orientado a objetos en Scala. En concreto, podéis elegir entre:

- la representación funcional de conjuntos basada en la noción matemática de funciones características (dificultad media).
- la representación de listas, similares a las proporcionadas por Scala, pero implementada por nosotros. En este caso se define la estructura de clases a usar (dificultad baja).
- la representación de árboles binarios. Se trata de la sección más abierta y que debéis definir de forma completa si optáis por esta alternativa. Cada nodo, interno u hoja, tendrá un identificador único. Los nodos hoja almacenarán valores de tipo **double** (dificultad alta).

Como se ha indicado, el objetivo de la práctica es poner en juego los conceptos de clases y objetos propios de este lenguaje de programación.

2 Conjuntos mediante funciones características

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.

```
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 como argumento un valor entero y devuelve un valor booleano indicando pertenencia o no).

Se deben implementar los siguientes métodos (queda a vuestra elección el lugar y forma más adecuada para implementarlos):

- **apply**, que recibe como argumento un valor entero e indica si este valor pertenece o no al conjunto.
- **toString**: ofrece una visión del contenido del conjunto. Para visualizar el conjunto se asume que se itera sobre un rango de valores dado por una constante llamada **LIMITE** (desde -LIMITE hasta +LIMITE) y se muestran aquellos que pertenecen al conjunto.
- **conjuntoUnElemento**: creación de un conjunto dado por un único elemento.
- **union**: dados dos objetos de la clase **Conjunto** produce su unión.
- **intersección**: intersección de dos objetos.
- **diferencia**: diferencia de dos objetos (el conjunto resultante está formado por aquellos valores que pertenecen al primer conjunto, pero no al segundo).
- **filtrar**: dado un conjunto y una función tipo $Int => Boolean$, devuelve como resultado un conjunto con los elementos que cumplen la condición indicada.
- **paraTodo**: comprueba si un determinado predicado se cumple para todos los elementos del conjunto. Esta función debe implementarse de forma recursiva, definiendo una función auxiliar, ya que hay que iterar sobre el rango de valores dado por LIMITE.
- **existe**: determina si un conjunto contiene al menos un elemento para el que se cumple un cierto predicado. Debe basarse en el método anterior.
- **map**: transforma un conjunto en otro aplicando una cierta función.

3 Clase Lista

La declaración de esta estructura debe basarse en la distinción entre una lista vacía y una lista con elementos. Usaremos la siguiente definición:

```scala
/**
  * Interfaz generica para la lista
  * @tparam A
  */
sealed trait Lista[+A]

/**
  * Objeto para definir lista vacia
  */
case object Nil extends Lista[Nothing]

/**
  * Clase para definir la lista como compuesta por elemento inicial
  * (cabeza) y resto (cola)
  * @param cabeza
  * @param cola
  * @tparam A
  */
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:

```scala
/**
 * Metodo para permitir crear listas sin usar new
 * @param elementos secuencia de elementos a incluir en la lista
 * @tparam A
 * @return
 */
def apply[A](elementos : A*) : Lista[A] = ???

/**
 * Obtiene la longitud de una lista
 * @param lista
 * @tparam A
 * @return
 */
def longitud[A](lista : Lista[A]) : Int = ???

/**
 * Metodo para sumar los valores de una lista de enteros
 * @param enteros
 * @return
 */
def sumaEnteros(enteros : Lista[Int]) : Double = ???

/**
 * Metodo para multiplicar los valores de una lista de enteros
 * @param enteros
 * @return
 */
```

```scala
def productoEnteros(enteros : Lista[Int]) : Double = ???

/**
 * Metodo para agregar el contenido de dos listas
 * @param lista1
 * @param lista2
 * @tparam A
 * @return
 */
def concatenar[A](lista1: Lista[A], lista2: Lista[A]): Lista[A] = ???

/**
 * Funcion de utilidad para aplicar una funcion de forma sucesiva a los
 * elementos de la lista con asociatividad por la derecha
 * @param lista
 * @param neutro
 * @param funcion
 * @tparam A
 * @tparam B
 * @return
*/
def foldRight[A, B](lista : Lista[A], neutro : B)(funcion : (A, B) => B): B = ???

/**
 * Suma mediante foldRight
 * @param listaEnteros
 * @return
 */
def sumaFoldRight(listaEnteros : Lista[Int]) : Double = ???

/**
 * Producto mediante foldRight
 * @param listaEnteros
 * @return
 */
def productoFoldRight(listaEnteros : Lista[Int]) : Double = ???

/**
 * Reemplaza la cabeza por nuevo valor. Se asume que si la lista esta vacia
 * se devuelve una lista con el nuevo elemento
 *
 * @param lista
 * @param cabezaNueva
 * @tparam A
 * @return
*/
def asignarCabeza[A](lista : Lista[A], cabezaNueva : A) : Lista[A] = ???

/**
 * Elimina el elemento cabeza de la lista
 * @param lista
 * @tparam A
 * @return
 */
```

```scala
def tail[A](lista : Lista[A]): Lista[A] = ???

/**
 * Elimina los n primeros elementos de una lista
 * @param lista lista con la que trabajar
 * @param n numero de elementos a eliminar
 * @tparam A tipo de datos
 * @return
 */
def eliminar[A](lista : Lista[A], n: Int) : Lista[A] = ???

/**
 * Elimina elementos mientra se cumple la condicion pasada como
 * argumento
 * @param lista lista con la que trabajar
 * @param criterio predicado a considerar para continuar con el borrado
 * @tparam A tipo de datos a usar
 * @return
 */
def eliminarMientras[A](lista : Lista[A], criterio: A => Boolean) : Lista[A] = ???


/**
 * Elimina el ultimo elemento de la lista. Aqui no se pueden compartir
 * datos en los objetos y hay que generar una nueva lista copiando
 * datos
 * @param lista lista con la que trabajar
 * @tparam A tipo de datos de la lista
 * @return
 */
def eliminarUltimo[A](lista : Lista[A]) : Lista[A] = ???

/**
 * foldLeft con recursividad tipo tail
 * @param lista lista con la que trabajar
 * @param neutro elemento neutro
 * @param funcion funcion a aplicar
 * @tparam A parametros de tipo de elementos de la lista
 * @tparam B parametro de tipo del elemento neutro
 * @return
 */
@annotation.tailrec
def foldLeft[A, B](lista : Lista[A], neutro: B)(funcion : (B, A) => B): B = ???
```

4 Árboles binarios

En este ejercicio debéis diseñar de forma completa la estructura de clases necesaria para representar árboles binarios. De igual forma, debéis definir un conjunto de operaciones mínimo que permita realizar recorridos en anchura y profundidad, determinar el tamaño del árbol (número de hojas y nodos internos), sumar los valores almacenados en las hojas, aplicar una función

a todas las hojas, generar un nuevo árbol a partir de otros dos así como cualquier otro que estimes necesario.

5 Observaciones

Al igual que en prácticas anteriores el código implementado debe superar un determinado conjunto de test de prueba que garanticen su correcto funcionamiento.

6 Defensa de la práctica y entrega del material

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. Además, se enviará correo a mgomez@decsai.ugr.es para concretar día y hora para la defensa. Se dispone hasta el día 20 de junio para realizar tanto la entrega como la defensa. El día del examen de la asignatura, 13 de Junio, estaré en el aula asignada para los que quieran hacer la defensa en dicho instante.