Práctica 5

códigos Huffman

Arquitectura general

En esta práctica he optado por la creación de un objeto que contendría las implementaciones de todos los métodos descritos en el guión para mantener de una forma encapsulada las funcionalidades de la práctica, un suit de testing para comprobar la funcionalidad de todas las funciones implementadas usando el entorno de Junit y Scalatest que hemos ido viendo a lo largo de la asignatura, creando test para comprobar la ejecución de funciones relativamente grandes de la práctica.

También he implementado una clase abstracta Nodo de la que extienden tanto la clase NodoHoja como NodoInterno, dos clases que se ocuparían de cumplir la funcionalidad de nodos en el árbol de codificación. Dichas clases contendrían únicamente los caracteres de dicho nodo y el contador que mostraría el peso del nodo. Y en el caso del NodoInterno dos nodos que se referirían a dos descendientes suyos.

Características del lenguaje

Entre muchas de las líneas de la práctica podemos destacar el uso de la programación funcional en el filtrado de listas, en el uso de foreach, map, filters, usos de case class, expresiones lambda etc.

Por remarcar algunas de las ubicaciones del código podríamos destacar:

```
def crearListaNodosHoja(listaCaracteresFrecuencia : List[(Char,Int)]):
List[NodoHoja] ={
  val resultado =  listaCaracteresFrecuencia.map(elemento =>
NodoHoja(elemento._1, elemento._2))

  resultado
}
```

Currying y funciones recursivas por la cola

```
@annotation.tailrec
def repetir(singleton: List[Nodo] => Boolean, combinar: List[Nodo] =>
List[Nodo])(listaNodos : List[Nodo]) :List[Nodo]={
  //Si la lista tiene un único elemento ya hemos terminado
  if(singleton(listaNodos)){
    listaNodos
  }else {//Si aún tiene más de un elemento combinaremos esos elementos en un
nodo intermedio.
    repetir(singleton,combinar)(combinar(listaNodos))
  }

}
```

Case class

```
def aux( mensajeSecreto: List[Int],nodo: Nodo): List[Char] ={
  nodo match {
    case NodoHoja(letra, _) =>
      if (mensajeSecreto.isEmpty) {
        List(letra)
      }else{
        letra :: aux(mensajeSecreto,raiz)
      }
    case NodoInterno(nodoIzquierda, nodoDerecha, _, _) =>
      if (mensajeSecreto.head == 0) {
        aux(mensajeSecreto.tail,nodoIzquierda )
      }
      else {
        aux(mensajeSecreto.tail, nodoDerecha)
      }
  }
```

Filtros

```
def codificarConTabla(tabla : TablaCodigo)(caracter : Char) : List[Int] ={
  //Se filtra en la tabla el caracter deseado y se extrae la lista de enteros
que corresponde a su codificación
  val result = tabla.filter(entrada => entrada._1 == caracter).head._2

  result
}
```

Y posiblemente algún que otro concepto no citado.

Versión

El entorno de desarrollo escogido ha sido IntelliJ IDEA 2016.3.5

La versión de Scala utilizada ha sido scala-sdk-2.11.6

Para el desarrollo de los test se ha incluido la librería de com.btmatthews.selenium.junit4:selenium-junit4-runner:1.0.4

y la librería au.com.dius:pact-jvm-provider-scalatest_2.11:3.3.6

Conclusión

En general la práctica me ha parecido una forma interesante de trabajar con varios de los conceptos tratados en clase sobre Scala y por ende sobre programación funcional. En general tanto esta práctica en concreto como las anteriores nos han servido para aprender el paradigma de la programación funcional y así poder aplicarla a algunas soluciones en nuestro futuro, como por ejemplo el tratamiento de datos en Android usando expresiones lambda, o bien conocer scala puede dotarnos de la capacidad de trabajar con fuentes de información bastante grandes con mucha facilidad.