Introducción a la Programación Funcional

Estructuras de datos recursivas

Universidad Autónoma de Entre Ríos

Estructuras de datos recursivas

- Una estructura de datos en Haskell o una clase en java o scala puede contener atributos esos atributos tienen un tipo determinado si ese tipo es igual a la clase contenedora tenemos una estructura recursiva.
- Una Lista por ejemplo puede verse como una estructura recursiva.

Estructuras de datos recursivas


```
 trait Lista[T] {
  def esVacio: Boolean
def primero:T
def resto:Lista[T]
}

class Vacia[T] extends Lista[T] {
def esVacio = true
def primero:Nothing = throw new NoSuchElementException("No existe elemento")
def resto:Nothing = throw new NoSuchElementException("No existe elemento")
}

class Llena[T](val primero: T, val resto: Lista[T]) extends Lista[T] {
   def esVacio= false
}
```

En Haskell :


```
data List a = Empty | Cons { listHead :: a, listTail :: List a} deriving (Show, Read, Eq, Ord)

ghci> Empty

Empty

ghci> 5 `Cons` Empty

Cons 5 Empty

ghci> 4 `Cons` (5 `Cons` Empty)

Cons 4 (Cons 5 Empty)

ghci> 3 `Cons` (4 `Cons` (5 `Cons` Empty))

Cons 3 (Cons 4 (Cons 5 Empty))
```

Case Classes

- Scala da soporte a la noción de clases caso (en inglés case classes, desde ahora clases Case). Las clases Case son clases regulares las cuales exportan sus parámetros constructores y a su vez proveen una descomposición recursiva de sí mismas a través de reconocimiento de patrones.
- A continuación se muestra un ejemplo para una jerarquía de clases la cual consiste de una super clase abstracta llamada Term y tres clases concretas: Var, Fun y App.

Case Classes

```
abstract class Term
case class Var(name: String) extends Term
case class Fun(arg: String, body: Term) extends Term
case class App(f: Term, v: Term) extends Term
```

Case Classes

```
trait Lista[T]

case class Vacia[T]() extends Lista[T]

case class Llena[T](val primero: T, val resto: Lista[T]) extends
Lista[T]
```

Pattern matching

- Pattern Matching nace del paradigma funcional aunque hoy en día lenguaje multiparadigma lo implementan como Scala o Kotlin. Pattern Matching permite definir funciones por medio de macheo de parámetros y resultados. Veamos un ejemplo en Haskell de definición de factorial:

```
factorial 0 = 1

factorial n = n * factorial (n - 1)
```

Erlang

```
fac(0) -> 1;
fac(N) when N > 0, is_integer(N) -> N * fac(N-1).
```

ML

```
fun fac 0 = 1
    | fac n = n * fac (n - 1)
```

Scala

```
def fact(n: Int): Int = n match {
    case 0 => 1
    case n => n * fact(n - 1)
 }
```

A la vez puedo utilizarlo con listas

```
def cantidad[T](xs:List[T]):Int = xs match {
 case List() => 0
 case head :: List()  => 1
 case head :: tail => 1 + cantidad(tail)
 }                                        //> cantidad: [T](xs: List[T])Int

 val l = List(1,3,4,5,6,8)                    //> l  : List[Int] = List(1, 3, 4, 5, 6, 8)

 cantidad(l)                              //> res0: Int = 6
```

Y como queda nuestra lista?

```
trait Lista[T]
case class Vacia[T]() extends Lista[T]
case class Llena[T](val primero: T, val resto: Lista[T]) extends Lista[T] {
}

def  esVacio[T](e:Lista[T]) = e match {
case e:Vacia[T] => true
case e:Llena[T] => false
}
```