Recursividad, Pattern Matching

Hernan Quintana (hquintan@ulima.edu.pe)

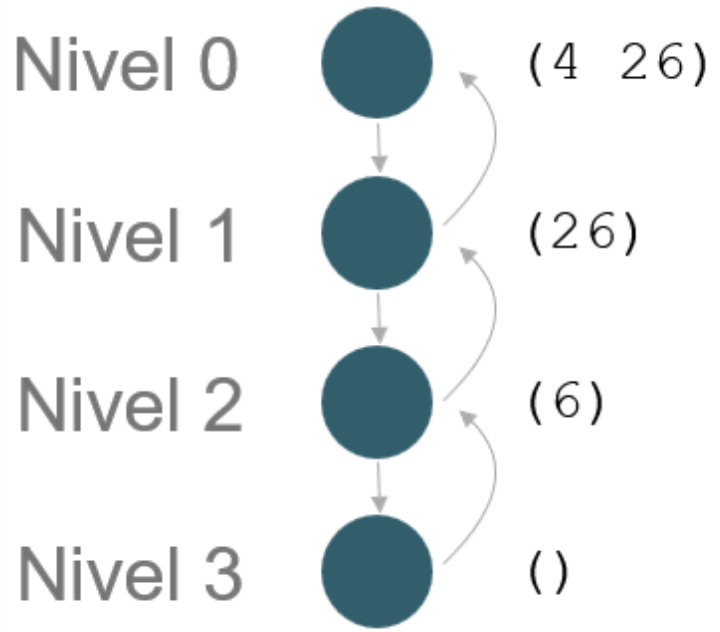
Recursividad

Loops

- Debido a que no hay asignaciones, tampoco se podrán realizar loops.
- Utilizaremos recursividad (llamar a la función que se está definiendo dentro de su propio cuerpo) para hacer "recorridos" de loops.

```
sumaTotalLista :: [Int] -> Int
sumaTotalLista lista =
    if null lista
        then 0
        else (head lista) + (sumaTotalLista (tail lista))
```

Recursion Tree



Ejercicio:

- Implementar una función llamada promedio, que calcule el promedio de una lista de números (Float).

Recursividad de cola

- https://www.youtube.com/watch?v=-PX0BV9hGZY

- Es una función cuya llamada recursiva, es la última en ejecutarse (evaluarse).
- Características:
  - La función debe definir un argumento/argumentos que vaya acumulando los resultados parciales.
  - En el caso recursivo, la función a ejecutarse al final (y escribirse primero), es la función recursiva.

Ejercicio

- Implementar una función llamada sumaListaTail, que calcule la suma de de números enteros, pero utilizando tail recursion.

```haskell
sumaListaTail :: [Int] -> Int -> Int
sumaListaTail lista acum =
    if null lista
        then acum
        else (sumaListaTail (tail lista) ((head lista) + acum))
```

Ejercicios

1. Implementar una función calcularPromedioTail utilizando Tail Recursion.
2. Implementar función mayorLista que calcule el número mayor de una lista.
3. Implementar una función que devuelva una lista en reversa.

Tuplas

Las tuplas son como listas pero con dos diferencias:

- Cantidad fija de elementos.
- Los elementos no necesitan ser necesariamente del mismo tipo de datos.

Ejemplos

```
(False, 1)
("Pepe", 30, 1.75)
((2,3), True)
((2,3), [2,3])
[(1,2), (3,4), (5,6)]
```

Definición de tipos Tupla

```
tuplaPersona :: (String, Int, Float) = ("Pepe", 30, 1.75)
```

Operaciones con tuplas

```
ghci> fst (1, 3)
ghci> 1
ghci> snd (1,3)
ghci> 3
```

Pattern Matching

Definición

- Característica que nos permite descomponer tipos de datos especificando los patrones de data que lo conforman.

Descomponer funciones

- Una función puede tener distintos cuerpos para diferentes patrones (argumentos de entrada).

```
obtenerEstadoCivil :: Int -> String
obtenerEstadoCivil 1 = "Soltero"
obtenerEstadoCivil 2 = "Casado"
obtenerEstadoCivil 3 = "Viudo"
obtenerEstadoCivil 4 = "Divorciado"
obtenerEstadoCivil x = "Otro"
```

Descomponer Listas

```haskell
sumList :: [Int] -> Int
sumList [] = 0   -- Patron para una lista vacia
sumList (x:xs) = x + sumList xs
```

Uso en tuplas

```
nombre :: (String, Int, Float) -> String
nombre (x, _, _) = x
```

Nombrando tipos

```
type Persona = (String, Int, Float)
nombre :: Persona -> String
nombre (x, _, _) = x
```

Guards

Otra manera de definir expresiones dado un predicado.

```haskell
categorizarEdad :: Int -> String
categorizarEdad  edad
    | edad < 12 = "Menor edad"
    | edad < 60 = "Adulto"
    | otherwise = "Adulto mayor"
```

List Comprehensions

- Expresiones que nos permiten armar sets más específicos de listas.
- Ejm: Crear una lista de los números pares entre hasta 10.

```
[x*2 | x <- [1..10]]
```

- Calcular la longitud de una lista de números, utilizando *list comprehension*.

```haskell
length' xs = sum [1 | _ <- xs]
```