

Programación Funcional en Haskell

Primera parte

Paradigmas de Lenguajes de Programación

Departamento de Ciencias de la Computación
Universidad de Buenos Aires

26 de marzo de 2024

Cómo empezar:

```
$ ghci
Loading ...
Prelude>:q
Leaving GHCi.
$ ghci test.hs
Loading ...
[1 of 1] Compiling Main ( test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

Otros comandos útiles:

- Para recargar: `:r`
- Para cargar otro archivo: `:l archivo.hs`
- Para conocer el tipo de una expresión: `:t True`

Currificación y aplicación parcial

```
prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> Int -> Int
prod' x y = x * y

prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> Int -> Int
prod' x y = x * y

prod :: (Int, Int) -> Int
prod (x, y) = x * y

prod' :: Int -> (Int -> Int)
(prod' x) y = x * y
```

¿Qué hacen estas funciones?

Currificación y aplicación parcial

Podría decirse que ambas “toman dos argumentos (x, y) y devuelven su producto”. Pero esto no es del todo así...

Las funciones en Haskell siempre toman un único argumento.

Entonces ¿qué hacen estas funciones?

- `prod` recibe una **tupla** de dos elementos.
- `prod'` es una función que **toma un x** de tipo `Int` y **devuelve una función** de tipo `Int -> Int`, cuyo comportamiento es tomar un entero y multiplicarlo por x .

En particular, `(prod' 2)` es la función que duplica.

Una definición equivalente de `prod'` usando funciones anónimas:

```
prod' x = \y -> x*y
```

Decimos que `prod'` es la versión **currificada** de `prod`.

Ejercicio

Definir las siguientes funciones:

- 1** `curry :: ((a,b) -> c) -> (a -> b -> c)`
que devuelve la versión currificada de una función no currificada.
- 2** `uncurry :: (a -> b -> c) -> ((a,b) -> c)`
que devuelve la versión no currificada de una función currificada.

Los paréntesis en gris no son necesarios, pero es útil escribirlos cuando estamos aprendiendo y queremos ver más explícitamente que estamos devolviendo una función.

Ejercicios

Sea la función:

```
prod :: Int -> Int -> Int  
prod x y = x * y
```

Definimos `doble x = prod 2 x`

- 1 ¿Cuál es el tipo de `doble`?
- 2 ¿Qué pasa si cambiamos la definición por `doble = prod 2`?
- 3 ¿Qué significa `(+) 1`?
- 4 Definir las siguientes funciones de forma similar a `(+) 1`:
 - `triple :: Float -> Float`
 - `esMayorDeEdad :: Int -> Bool`

Ejercicios

- 1** Implementar y dar los tipos de las siguientes funciones:
 - a** `(.)` que compone dos funciones. Por ejemplo:
`((\x -> x * 4).(\y -> y - 3)) 10` devuelve 28.
 - b** `flip` que invierte los argumentos de una función. Por ejemplo:
`flip (\x y -> x - y) 1 5` devuelve 4.
 - c** `(\$)` que aplica una función a un argumento. Por ejemplo:
`id \$ 6` devuelve 6.
 - d** `const` que, dado un valor, retorna una función constante que devuelve siempre ese valor. Por ejemplo:
`const 5 'casa'` devuelve 5.
- 2** ¿Qué hace `flip (\$) 0`?
- 3** ¿Y `(==0) . (flip mod 2)`?

Pueden ver más funciones útiles en la sección Útil del Campus.

Hay varias **macros** para definir listas:

- **Por extensión**

Esto es, dar la lista explícita, escribiendo todos sus elementos.
Por ejemplo: `[4, 3, 3, 4, 6, 5, 4, 5, 4, 5]`.

- **Secuencias**

Son progresiones aritméticas en un rango particular.
Por ejemplo: `[3..7]` es la lista que tiene todos los números enteros entre 3 y 7, mientras que `[2, 5..18]` es la lista que contiene 2, 5, 8, 11, 14 y 17.

- **Por comprensión**

Se definen de la siguiente manera:

`[expresión | selectores, condiciones]`

Por ejemplo: `[(x,y) | x <- [0..5], y <- [0..3], x+y==4]`
es la lista que tiene los pares (1,3), (2,2), (3,1) y (4,0).

Haskell también nos permite trabajar con **listas infinitas**.

Algunos ejemplos:

- `naturales = [1..]`

1, 2, 3, 4, ...

- `multiplosDe3 = [0,3..]`

0, 3, 6, 9, ...

- `repeat 'hola'`

"hola", "hola", "hola", "hola", ...

- `primos = [n | n <- [2..], esPrimo n]`

(asumiendo `esPrimo` definida) 2, 3, 5, 7, ...

- `infinitosUnos = 1 : infinitosUnos`

1, 1, 1, 1, ...

¿Cómo es posible trabajar con listas infinitas sin que se cuelgue?

```
take :: Int -> [a] -> [a]
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs

infinitosUnos :: [Int]
infinitosUnos = 1 : infinitosUnos

nUnos :: Int -> [Int]
nUnos n = take n infinitosUnos
```

- Si ejecutamos `nUnos 2`...

```
nUnos 2 → take 2 infinitosUnos → take 2 (1:infinitosUnos) →
1 : take (2-1) infinitosUnos → 1 : take 1 infinitosUnos →
1 : 1:take (1-1) infinitosUnos → 1 : take 0 infinitosUnos →
1 : 1 : []
```

- ¿Qué sucedería si usáramos otra estrategia de reducción?
- Si para algún término existe una reducción finita, entonces la estrategia de reducción lazy termina.

Funciones de alto orden

Definamos las siguientes funciones

Precondición: las listas tienen algún elemento.

- `maximo :: Ord a => [a] -> a`
- `minimo :: Ord a => [a] -> a`
- `listaMasCorta :: [[a]] -> [a]`

Siempre hago lo mismo... ¿Se podrá generalizar? ¿Cómo?

Ejercicio

- `mejorSegun :: (a -> a -> Bool) -> [a] -> a`
- Reescribir `maximo` y `listaMasCorta` en base a `mejorSegun`

Esquemas de recursión sobre listas: filter

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs) =
    if p x
    then x : filter p xs
    else filter p xs
```

Ejercicios

Definir usando filter:

- 1 deLongitudN :: Int -> [[a]] -> [[a]]
- 2 soloPuntosFijosEnN :: Int -> [Int->Int] -> [Int->Int]

Dados un número n y una lista de funciones, deja las funciones que al aplicarlas a n dan n .

Esquemas de recursión sobre listas: map

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Ejercicio

Definir usando map:

- 1** `reverseAnidado :: [[Char]] -> [[Char]]` que, dada una lista de strings, devuelve una lista con cada string dado vuelta y la lista completa dada vuelta. Por ejemplo: `reverseAnidado [“quedate”, “en”, “casa”]` devuelve [“asac”, “ne”, “etadeuq”].
Ayuda: ya existe la función `reverse` que invierte una lista.
- 2** `paresCuadrados :: [Int] -> [Int]` que, dada una lista de enteros, devuelve una lista con los cuadrados de los números pares, y los impares sin modificar.

Desplegando la macro de las listas por comprensión

Definir una expresión equivalente a las siguiente utilizando `map` y `filter`:

Ejercicio

```
listaComp f xs p = [f x | x <- xs, p x]
```

Nota: `concatMap = concat . map`

Esquemas de recursión estructural sobre listas

Ya conocen `foldr` y `foldl`.

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`.

Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía, y el tipo del resultado debe ser el de los elementos de la lista.

Ejercicio

Definir `mejorSegún :: (a -> a -> Bool) -> [a] -> a` usando `foldr1` o `foldl1`.

Implementar las siguientes funciones utilizando esquemas de recursión

- 1** `elem :: Eq a => a -> [a] -> Bool` que indica si un elemento pertenece o no a la lista.
- 2** `sumaAlt`, que realiza la suma alternada de los elementos de una lista. Es decir, da como resultado: el primer elemento, menos el segundo, más el tercero, menos el cuarto, etc.
- 3** `sacarPrimera :: Eq a => a -> [a] -> [a]` que elimina la primera aparición de un elemento en la lista.

¿Qué otros esquemas de recursión conocen?

Sea el siguiente tipo:

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)
```

```
Ejemplo: miÁrbol = Bin (Hoja 3) 5 (Bin (Hoja 7) 8 (Hoja 1))
```

Definir el esquema de recursión estructural (*fold*) para árboles estrictamente binarios, y dar su tipo.

El esquema debe permitir definir las funciones altura, ramas, #nodos, #hojas, espejo, etc.

¿Cómo hacemos?

Recordemos el tipo de `foldr`, el esquema de recursión estructural para listas.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

¿Por qué tiene ese tipo?

(Pista: pensar en cuáles son los constructores del tipo `[a]`).

Un esquema de recursión estructural espera recibir **un argumento por cada constructor** (para saber qué devolver en cada caso), y además **la estructura que va a recorrer**.

El tipo de cada argumento va a depender de lo que reciba el constructor correspondiente. (¡Y todos van a devolver lo mismo!)

Si el constructor es recursivo, el argumento correspondiente del `fold` va a recibir el resultado de cada llamada recursiva.

¿Cómo hacemos? (Continúa)

Miremos bien la estructura del tipo.

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)
```

Estamos ante un tipo inductivo con un constructor *no recursivo* y un constructor *recursivo*.

¿Cuál va a ser el tipo de nuestro fold?

¿Y la implementación?

```
foldAEB :: (a -> b) -> (b -> a -> b -> b) -> AEB a -> b
foldAEB fHoja fBin t  =  case t of
    Hoja n             -> fHoja n
    Bin t1 n t2        -> fBin (rec t1) n (rec t2)
                        where rec = foldAEB fHoja fBin
```

Ejercicio para ustedes: definir las funciones altura, ramas, #nodos, #hojas y espejo usando foldAEB.

Si quieren podemos hacer alguna en el pizarrón.

Dado el siguiente tipo que representa polinomios:

```
data Polinomio a = X
                  | Cte a
                  | Suma (Polinomio a) (Polinomio a)
                  | Prod (Polinomio a) (Polinomio a)
```

- Definir la función
evaluar :: Num a => a -> Polinomio a -> a
- Definir el esquema de recursión estructural foldPoli para polinomios (y dar su tipo).
- Redefinir evaluar usando foldPoli.

Una estructura más compleja

Dado el tipo de datos

```
data RoseTree a = Rose a [RoseTree a]
```

de árboles donde cada nodo tiene una cantidad indeterminada de hijos.

Escribir alguna de las siguientes funciones:

- `hojas`, que dado un `RoseTree`, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
- `ramas`, que dado un `RoseTree`, devuelva los caminos de su raíz a cada una de sus hojas.
- `tamaño`, que devuelve la cantidad de nodos de un `RoseTree`.
- `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.

Una estructura más compleja

Dado el tipo de datos

```
data RoseTree a = Rose a [RoseTree a]
```

de árboles donde cada nodo tiene una cantidad indeterminada de hijos.

- 1 Escribir el esquema de recursión estructural para `RoseTree`.
- 2 Usando el esquema definido, escribir las siguientes funciones:
 - `hojas`, que dado un `RoseTree`, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
 - `ramas`, que dado un `RoseTree`, devuelva los caminos de su raíz a cada una de sus hojas.
 - `tamaño`, que devuelve la cantidad de nodos de un `RoseTree`.
 - `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.

i i i i i i i i i i ? ? ? ? ? ? ? ?