

Lenguajes de programación

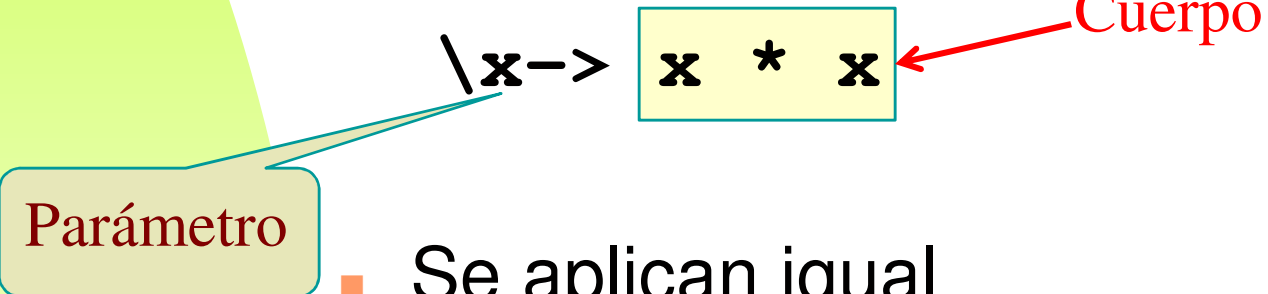
Funciones Lambda y Funciones de Orden Superior en Haskell

Lambdas en Haskell

- Funcionan igual que en Scheme

- Se escriben como:

`\x -> x * x`



Parámetro

- Se aplican igual

`(\x -> x * x) 5 ==> 25`

Ejercicio

- Hacer una expresión lambda que multiplique los componentes de un par:

$(\backslash x \rightarrow \dots) (2, 3) \Rightarrow 6$

👉 **Hint:** la función **fst** da el primer elemento de una tupla, **snd** da el segundo, o sea,

fst (2, 3) \Rightarrow 2

snd (2, 3) \Rightarrow 3

Ejemplo voltea

Función `flip`
de Haskell

- La función `voltea` invierte el orden de los argumentos de una función binaria

`voltea div 6 2 ==> 0`

`div 6 2 ==> 3`

- Tipo de dato:

`voltea :: (a -> b -> c) ->
 b -> a -> c`

- Solución:

`voltea = \f x y -> f y x`

Funciones de orden superior

- Son funciones que se aplican a otras funciones
- Haskell tiene las FOS predefinidas
 - ◆ `map`
 - ◆ `all, any`
 - ◆ `filter`
 - ◆ `foldl, foldl1`
 - ◆ `foldr, foldr1`
 - ◆ `.` (`compose`)
 - ◆ `until`
 - ◆ ...

map

- La función **map** aplica una función a todos los elementos de una lista y regresa el resultado en otra lista
 - ◆ Recibe una función $f: a \rightarrow b$
 - ◆ Recibe otro argumento que es una lista de a
 - ◆ Regresa como resultado una lista de b

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

```
map sqrt [1, 2, 3, 4, 5]
```

```
=> [1, 1.41421, 1.73205, 2, 2.23607]
```

Ejemplo

- Obtiene el producto punto de dos arreglos dados como listas

```
producto [1,2,3] [4,5,6] => 32
```

```
producto :: [Int] -> [Int] -> Int
```

```
producto l1 l2 =
```

```
    sum (map (\(p,s) -> p * s)
```

```
        (zip l1 l2))
```

Regresa una lista de
pares correspondientes

all, any

- La función **all** aplica un predicado a todos los elementos de una lista y determina si todos lo cumplen

```
all :: (a -> Bool) -> [a] -> Bool
```

```
all even [1,2,3,4,5] => False
```

Determina
si es par

- La función **any** aplica un predicado a todos los elementos de una lista y determina si alguno lo cumple

```
any :: (a -> Bool) -> [a] -> Bool
```

```
any even [1,2,3,4,5] => True
```


filter

- La función **filter** genera una lista con los elementos que cumplan el predicado que se recibe como parámetro.

```
filter :: (a -> Bool) -> [a]  
      -> [a]
```

```
filter even [1,2,3,4,5]  
=> [2,4]
```

```
filter odd [1,2,3,4,5]  
=> [1,3,5]
```

foldl, foldl1

- La función `foldl` reduce una lista aplicando un operador binario (primer argumento) de izquierda a derecha, y utilizando como primer operando su segundo argumento. `foldl1` hace lo mismo pero sin valor inicial.

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

```
foldl f z [] = z
```

```
foldl f z (x:xs) = foldl f (f z x) xs
```

```
foldl1 :: (a -> a -> a) -> [a] -> a
```

```
foldl1 f (x:xs) = foldl f x xs
```

- Ejemplos:

```
foldl (-) 1 [4,2,3] => 1-4-2-3 = -8
```

```
foldl1 (/) [1,2,3] => 1/2/3 = 0.1667
```

Ejemplo

- Invierte el orden de los elementos de una lista usando foldl

```
invierte = foldl (flip (:)) []
```

```
invierte [1,2,3]
```

```
=> foldl (flip (:)) [] [1,2,3]
```

```
=> foldl (flip (:)) ((flip (:)) [] 1) [2,3]
```

```
=> foldl (flip (:)) [1] [2,3]
```

```
=> foldl (flip (:)) ((flip (:)) [1] 2) [3]
```

```
=> foldl (flip (:)) [2,1] [3]
```

```
=> foldl (flip (:)) ((flip (:)) [1,2] 3) []
```

```
=> foldl (flip (:)) [3,2,1] []
```

```
=> [3,2,1]
```

foldr, foldr1

- Las funciones `foldr`, y `foldr1` son los duales de derecha a izquierda de `foldl` y `foldl1`, respectivamente.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs) = f x (foldr f z xs)
```

```
foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      = x
foldr1 f (x:xs)   = f x (foldr1 f xs)
```

- Ejemplos:

```
foldr (-) 1 [4,2,3] => 4-(2-(3-1)) = 4
```

```
foldr1 (/) [8,4,2] => 8/(4/2) = 4.0
```

Compose (.)

- Toma dos funciones y las “encadena”

- ◆ $(f . g . h) x$ es equivalente a $f (g (h x))$

- ◆ Por ejemplo:

$(\text{sqrt} . \text{sum}) \ [1, 2, 3, 4, 5]$

$\Rightarrow \ 3.87298$

- Tipo de datos:

$(.) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

$f . g = \ \backslash \ x \rightarrow f (g x)$

Ejemplo

- **sumabs** calcula la suma de los valores absolutos de los elementos de una lista

sumabs [-1, -2, -3] => 6

sumabs = **sum** . **map abs**

- ◆ La definición de **sumabs** es sin argumentos

until

- “until” toma un predicado, una función unaria y un valor, y reaplica la función al valor hasta que cumpla el predicado

- Ejemplos:

```
until (> 5) (+ 2) 0 => 6
```

```
until (> 1000) (* 2) 1 => 1024
```

```
= until (\x->x > 1000) (\x->x * 2) 1
```

- Tipo de dato:

```
until :: (a -> Bool) -> (a -> a)  
      -> a -> a
```