Tema 4. Funciones sobre listas

Funciones predefinidas:	Ejemplos:
head :: [α] –> α	head [2,4,6,1] = 2
tail :: [α] –> [α]	tail [2,4,6,1] = [4,6,1]
last :: [α] –> α	last [2,4,6,1] = 1
init :: [α] –> [α]	init [2,4,6,1] = [2,4,6]
length :: [α] –> Int	length [2,4,6,1] = 4
null :: [α] –> Bool	null [2,4,6,1] = False
reverse :: [α] –> [α]	reverse [2,4,6,1] = [1,6,4,2]

Operador de concatenación (++) :: [α] –> [α] –> [α]

Ej: [8,9,1,5] ++ [7,1,2,6] = [8,9,1,5,7,1,2,6]

Funciones sobre listas (2)

take, drop :: Int –> [α] –> [α]

take 2 [8,9,1,5,4] = [8,9]   drop 2 [8,9,1,5,4] = [1,5,4]

take 0 [8,9,1,5] = []   drop 0 [8,9,1,5] = [8,9,1,5]

splitAt :: Int –> [α] –> ([α],[α])

splitAt 4 [8,9,1,5,3,7,8] = ([8,9,1,5],[3,7,8])

Propiedades:

splitAt n s = (take n s, drop n s)

take n s ++ drop n s = s

length (s1 ++ s2) = length s1 + length s2   ... etc

Funciones sobre listas (3)

? zip [0..2] "epa" zip :: [α] –> [β] –> [(α,β)]

[(0,'e'),(1,'p'),(2,'a')]

zip3 :: [α] –> [β] –> [γ] –>[(α,β,γ)]

? zip3 "abcd" [1,2,3] ["p","rue","ba"]

[('a',1,"p"),('b',2,"rue"),('c',3,"ba")]

unzip :: [(α,β)] –> ([α],[β])

? unzip [(4,'e'),(2,'p'),(5,'a')]

([4,2,5], "epa")

unzip3 :: [(α,β,γ)] –> ([α],[β],[γ])

? unzip3 [(0,12,23),(3,24,7)]

([0,3],[12,24],[23,7])

Orden superior.

Función de O. Superior: es una función con algún argumento de tipo funcional a su vez.

<u>Ej</u>: Las funciones vistas curry y uncurry son de orden superior

curry f x y = f (x,y) tiene tipo:

$$\text{curry} :: ((\alpha,\beta) \to \gamma) \to \alpha \to \beta \to \gamma$$

uncurry g (x,y) = g x y tiene tipo:

$$\text{uncurry} :: (\alpha \to \beta \to \gamma) \to (\alpha,\beta) \to \gamma$$

Función filter

filter :: (α –> Bool) –> [α] –> [α]

Definición recursiva:

filter p [] = []

filter p (x:s) = if p x then x:filter p s else filter p s

Ejemplos:

? let par x = (mod x 2 == 0) in filter par [4,1,3,6,5]

[4,6]

? filter isAlpha [‘d’,‘+’,‘S’,’8’, ‘#’, ‘x’]

[‘d’,‘S’,‘x’] (== “dSx”)

Función map

map :: (α –> β) –> [α] –> [β]

Definición recursiva:

map f [] = []

map f (x:s) = f x : map f s

Ejemplos:

? let doble x = 2∗x in map doble [4,1,3,5]

[8,2,6,10]

? let lis = map (sumdo 3) [4,5,2]; sumdo x y = x + 2*y in lis

[11,13,7]

Ejemplos de map y filter (1)

- map y filter son "esquemas recursivos" generales
- Otras funciones se pueden definir mediante ellos

Ejemplo   Definir   dobPares :: [Int] –> [Int] tal que

? dobPares [4,1,3,6,5]

[8,12]

Solución 1: definición recursiva directa

dobpares [] = []

dobpares (x:s) = if par x then 2*x : dobPares s else dobPares s

Ejemplos de map y filter (2)

Solución 2: Definir dobPares usando map y filter

dobPares s = map (2∗) (filter par s)

ó equivalentemente:

dobPares = (map (2∗)) . (filter par)

? dobPares [4,1,3,6,5]

[8,12]

$$
\begin{array}{rc}
 & [4,1,3,6,5] \\
\text{filter par} & \downarrow \\
 & [4,6] \\
\text{map } (2*) & \downarrow \\
 & [8,12]
\end{array}
$$

Razonamiento formal

Ejemplo Definir posPares :: [Int] –> [Int] tal que

? posPares [-4,1,6,-3,8,2]

[6,8,2]

Soluciones: posPares = (filter (>0)) . (filter par)

posPares' = (filter par) . (filter (>0))

¿Son equivalentes posPares y posPares'?

¿Es cierta en general la propiedad siguiente?

$$\boxed{\text{(filter p) . (filter q) = (filter q) . (filter p)}}$$

Demostración por inducción

Probar formalmente el "teorema" siguiente:

∀p ∀q ∀s , filter p (filter q s) = filter q (filter p s)

Demostración:

- caso simple (s = [])

  ¿ filter p (filter q []) = filter q (filter p []) ?

- hipótesis de inducción: cierto el teorema para s
- caso general (x:s)

  ¿ filter p (filter q (x:s)) = filter q (filter p (x:s)) ?

se puede usar la h.i.

Funciones foldl y foldr

Esquemas fold ("aplanar") por izquierda (l) y derecha (r)

Idea $\otimes$ op. binario, e semilla

foldl ($\otimes$) e [x_1, x_2, x_3, x_4]

$$= (((e \otimes x_1) \otimes x_2) \otimes x_3) \otimes x_4$$

foldlr ($\otimes$) e [x_1, x_2, x_3, x_4]

$$= x_1 \otimes (x_2 \otimes (x_3 \otimes (x_4 \otimes e)))$$

Ejemplos

? foldl (+) 0 [8,2,6,10]
26

? foldr (*) 1 [3,2,5,10]
300

Ejemplos de uso de foldr

Funciones predefinidas que usan foldr

- sum = foldr (+) 0
- product = foldr (*) 1
- and = foldr (&&) True
- or = foldr (| |) False
- concat = foldr (++) []

Ejemplos

? product [3,2,5,10] => 300

? and [True, False, True] => False

? concat [[3,2],[5],[8,1,7]] => [3,2,5,8,1,7]

Tipo y definición de foldl y foldr

foldl :: (α –> β –> α) –> α –> [β] –> α

foldl f e [] = e

foldl f e (x:s) = foldl f (f e x) s

foldr :: (α –> β –> β) –> β –> [α] –> β

foldr f e [] = e

foldr f e (x:s) = f x (foldr f e s)

Propiedad (teorema de dualidad):

Si (⊗) es asociativa y e es el neutro de ⊗ entonces

foldl (⊗) e = foldr (⊗) e

Variantes foldl1 y foldr1

Para aplanar (sin semilla e), sobre listas no vacías

foldl1, foldr1 :: $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow \alpha$

foldl1 $(\otimes)$ $[x_1, x_2,\ldots, x_n] = (\ldots((x_1 \otimes x_2) \otimes x_3) \ldots) \otimes x_n$

foldr1 $(\otimes)$ $[x_1, x_2,\ldots, x_n] = x_1 \otimes (x_2 \otimes (\ldots (x_{n-1} \otimes x_n)\ldots))$

Funciones predefinidas que usan foldl1:

- maximum = foldl1 max
- minimum = foldl1 min

Ejemplos

? maximum [3,2,5,10,4] => 10

? minimum ["hola","como","estas"] => "como"

Funciones takeWhile, dropWhile, span

Ejemplos de uso

? takeWhile par [8,2,1,6,3,10] => [8,2]

? dropWhile par [8,2,1,6,3,10] => [1,6,3,10]

? span par [8,2,1,6,3,10] => ([8,2], [1,6,3,10])

? takeWhile (/= ' ') "hola que tal te va" => "hola"

? dropWhile (/= ' ') "hola que tal te va" => " que tal te va"

Propiedades

takeWhile p s ++ dropWhile p s = s

span p s = (takeWhile p s, dropWhile p s)

Tipo y definición de takeWhile, dropWhile

takeWhile, dropWhile :: (α–>Bool) –>[α] –>[α]

```
takeWhile p  [] = []
takeWhile p (x:s)
        | p x           = x: takeWhile p s
        | otherwise     = []

dropWhile p  [] = []
dropWhile p (x:s)
        | p x           = dropWhile p s
        | otherwise     = (x:s)
```

Ejemplo de uso de takeWhile, dropWhile

Ejercicio: Obtener la lista de palabras de una frase dada

Definir lisPal :: String –> [String] de modo que:

? lisPal " Esto   es una prueba "

[“Esto”, “es”, “una”, “prueba”]

? lisPal " hola   "

[“hola”]

? lisPal " "

[]

Operador de composición (1)

$(.) :: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \gamma$

(.) f g x = f (g x)

Aplicación parcial (a dos argumentos) nos da:

$$f \,.\, g \;\equiv\; (.)\; f\; g \;\equiv\; \lambda x.\, f\,(g\; x)$$

Se puede definir funciones como composición de otras

Ej:

dobPares = (map doble) **.** (filter par)

cuyo tipo se obtiene de la siguiente forma:

Operador de composición (2)

dobPares = (map doble) **.** (filter par)

doble :: Int –> Int

map :: (α –> β) –> [α] –> [β]

map doble :: [Int] –> [Int]

par :: Int –> Bool

filter :: (α –> Bool) –> [α] –> [α]

filter par :: [Int] –> [Int]

Como **(.)** :: (β –> γ) –> (α –> β) –> α –> γ

se obtiene: **dobPares :: [Int] –> [Int]**

Función error y notación λ

error :: String –> α

Es polimórfica para ser usada con todos los tipos. Ejs:

factorial x = if x < 0 then error "dato negativo" else ....

factorial :: Int –> Int   error :: String –> Int

sigLetra x = if x=='z' then error "siguiente de z" else ....

sigLetra :: Char –> Char   error :: String –> Char

Notación λ para funciones anónimas λx. exp ≡ \x –> exp

Ej: map (\x –> x+1) [4,5,6]

[5,6,7]

Listas intensionales (1)

Notación alternativa de listas para cálculos con map y filter

[<expresión> | <cualif>$_1$, <cualif>$_2$, ..., <cualif>$_n$]

donde cada <cualif>$_i$ puede ser:

- generador <patrón> <– <expr-lista>
- filtro <expr-booleana>

Ejemplos:

? [x*x | x <– [1..7]] => [1,4,9,16,25,36,49]

$$? [x*x \mid \underbrace{x <– [1..7]}_{\text{generador}}, \underbrace{\text{mod } x\ 2 == 1}_{\text{filtro}}] \quad => \quad [1,9,25,49]$$

Listas intensionales (2)

Combinación de generadores y filtros:

```
? [ (x,y) | x <– [1..3], y<– [5..6] ]
  [(1,5),(1,6),(2,5),(2,6),(3,5),(3,6)]
```

"anidamiento"

```
? [ (i,j) | i <– [1..4], j<– [i+1..4] ]
  [(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)]
```

"j depende de i"

```
? [ (i,j) | i <– [1..4], even i, j<– [i+1..4], j>i+1]
  [(2,4)]
```

generadores filtros

```
? [3 | j<– [1..4]]
  [3,3,3,3]
```

Listas intensionales (3)

Definición de funciones mediante listas intensionales:

blancos :: Int –> String

blancos n = [‘ ’ | i <– [1..n]]

divisores :: Int –> [Int]

divisores x = [z | z <– [1.. div x 2], mod x z == 0] ++ [x]

esprimo :: Int –> Bool

esprimo n = (divisores n == [1,n])

listaPrimos :: [Int]

listaPrimos = [n | n<–[2..], esprimo n]

Strings

type String = [Char]   “hola” = [‘h’,’o’,’l’,’a’]

- *Tipo especial para imprimir en pantalla mediante*

putStr :: String –> IO ()   -- primitiva

Ejs: ? “hola”   ? putStr “hola”

“hola”   hola

? “hola” ++ “\n” ++ blancos 2 ++ “adios”

“hola\n   adios”

? putStr (“hola” ++ “\n” ++ blancos 2 ++ “adios”)

hola

  adios