Listas por comprensión

Dpto. Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

Recordatorio de listas

Recordemos que una lista es una secuencia ordenada de elementos del mismo tipo en la que se permiten las repeticiones.

- Declaración del tipo: `lista :: [Tipo]` siendo `Tipo` el tipo de los elementos que componen `lista`

```
[] :: [a]
[True, True, False] :: [Bool]
```

- Las expresiones se construyen con `[]` (la lista vacía), `x:xs` (la lista que tiene como primer elemento `x` y restantes elementos la lista `xs`) o enumerando entre corchetes y separados por coma los elementos que la componen.

```
'a':('b':[]) :: String
[[1], [1, 2, 3]] :: [[Int]]
```

Enumeración de listas

Podemos usar la **enumeración** de listas para generar listas de forma sencilla. El tipo debe pertenecer a la clase Enum.

- `[x..y]`, son los elementos desde x hasta y. Por defecto se van añadiendo los elementos consecutivos. Por ejemplo

```
[1..10] == [1,2,3,4,5,6,7,8,9,10]
['a'..'c'] == ['a','b','c'] == "abc"
```

- Para poder dar saltos, hay que ayudarle indicando el primer elemento y el siguiente: `[x,y..z]`. De esta forma, se calcula la diferencia entre `x` e `y` para los saltos, sin sobrepasar `z`. Por ejemplo:

```
[1,3..9] == [1,3,5,7,9]
[1,3..10] == [1,3,5,7,9]
[2.2..5] == [2.2,3.2,4.2]
[1,1.5..2.7] == [1.0,1.5,2.0,2.5]
[5,4..1] == [5,4,3,2,1]  -- también vale para un orden inverso
```

Funciones sobre listas (I)

- `!!`: tomar un elemento de un lista.

```
[5,7,8,6] !! 2 == 8
```

- `head / last`: primer / último elemento de una lista.

```
head [5,7,8,6] == 5
last [5,7,8,6] == 6
```

- `take / drop / tail`: coger / eliminar *n* elementos de una lista.

```
take 3 [5,7,8,6] == [5,7,8]
drop 3 [5,7,8,6] == [6]
tail [5,7,8,6] == [7,8,6] == drop 1 [5,7,8,6]
init [5,7,8,6] == [5,7,8]
```

- `reverse`: invierte una lista.

```
reverse [5,7,8,6] == [6,5,7,5]
```

Funciones sobre listas (II)

- `concat`: concatena una lista de listas.

```
concat [[1],[2,3,4],[5,6]] = [1,2,3,4,5,6]
```

- `elem / notElem`: determina si un elemento está o no en una lista.

```
elem 3 [5,7,8,6] == False
elem 7 [5,7,8,6] == True
notElem 3 [5,7,8,6] == False
notElem 7 [5,7,8,6] == True
```

- `takeWhile / dropWhile`: toma o elimina elementos de una lista hasta que se deje de cumplir una condición dada.

```
takeWhile (<5) [1,2,3,4,5,6,7,8] == [1,2]
dropWhile (<5) [1,2,3,4,5,6,7,8] == [3,4,5,6,7,8]
```

La función zip

- (`zip xs ys`) es la lista obtenida emparejando los elementos de las listas xs e ys. Por ejemplo,

```
> zip ['a','b','c'] [2,5,4]
[('a',2),('b',5),('c',4)]
```

- La longitud de la lista obtenida es igual al de la lista más corta.

```
> zip "Haskell" [2,4..100]
[('H',2),('a',4),('s',6),('k',8),('e',10),('l',12),('l',14)]
```

- Un ejemplo muy útil: definición de función que asocia a cada elemento su posición en la lista:

```
> posiciones xs = zip xs [1..length xs]
> posiciones "Hash"
[('H',1),('a',2),('s',3),('h',4)]
```

Listas por comprensión

Tienen la forma `[expresión | generador+, guarda*]`, donde `generador` tiene la forma `var <- lista` y `guarda` es una condición.
Así, si en Matemáticas tenemos
$\{x^2 : x \in \{2, 3, 4, 5\}\} = \{4, 9, 16, 25\}$, en Haskell se traduce en:

```
> [x^2 | x <- [2..5]]
[4,9,16,25]
```

Otros ejemplos:

```
> [2 * x | x <- [1..10]]
[2,4,6,8,10,12,14,16,18,20]
> [even x | x <- [1..10]]
[False,True,False,True,False,True,False,True,False,True]
> [mod x 5 | x <- [0..15]]
[0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0]
```

Listas por comprensión

Generadores

La expresión `x <- [2..5]` se denomina `generador`. Se puede emplear más de un generador en una lista por comprensión:

```
> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Además, unos generadores pueden depender de otros:

```
>  [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Definición de `concat`:

```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]

>  concat [[1,3],[2,5,6],[4,7]]
[1,3,2,5,6,4,7]
```

Listas por comprensión

Generadores con variables anónimas

Ejemplo de generador con variable anónima: `(primeros ps)` es la lista de los primeros elementos de la lista de pares ps.

```
primeros :: [(a, b)] -> [a]
primeros ps = [x | (x,_) <- ps]

> primeros [(1,3),(2,5),(6,3)]
[1,2,6]
```

Definición de la longitud por comprensión:

```
length :: [a] -> Int
length xs = sum [1 | _ <- xs]
```

Guardas

- Las listas por comprensión pueden tener guardas para restringir los valores.
- Ejemplo de guarda (`even x`):

```
> primeros  [x | x <- [1..10], even x]
[2,4,6,8,10]
```

Guardas

Guarda con igualdad

- Una lista de asociación es una lista de pares formado por una clave y un valor. Por ejemplo, una lista de puntos obtenidos en un juego asociados a nombres:

```
[("Juan",2),("Ana",3),("Eva",1),("Juan",5)]
```

- (puntos n xs) es la lista de los valores de la lista de asociación ds cuyas claves valen n.

```
puntos :: Eq a => a -> [(a, b)] -> [b]
puntos n xs = [p | (na, p) <- xs, na == n]

> puntos "Juan" [("Juan",2),("Ana",3),("Eva",1),("Juan",5)]
[2,5]
```

Listas por comprensión

Ejemplos:

```
> [(x, y) | x <- [1,3], y <- "par"]
```

Listas por comprensión

Ejemplos:

```
> [(x, y) | x <- [1,3], y <- "par"]
[(1,'p'),(1,'a'),(1,'r'),(3,'p'),(3,'a'),(3,'r')]
```

```
> [y | x <- [2,4..8], y <- [x..8]]
```

Listas por comprensión

Ejemplos:

```
> [(x, y) | x <- [1,3], y <- "par"]
[(1,'p'),(1,'a'),(1,'r'),(3,'p'),(3,'a'),(3,'r')]
```

```
> [y | x <- [2,4..8], y <- [x..8]]
[2,3,4,5,6,7,8,4,5,6,7,8,6,7,8,8]
```

```
> [ (x:[y..4]) | x <- [1..4], y <- [1..4], y > x]
```

Listas por comprensión

Ejemplos:

```
> [(x, y) | x <- [1,3], y <- "par"]
[(1,'p'),(1,'a'),(1,'r'),(3,'p'),(3,'a'),(3,'r')]
```

```
> [y | x <- [2,4..8], y <- [x..8]]
[2,3,4,5,6,7,8,4,5,6,7,8,6,7,8,8]
```

```
> [ (x:[y..4]) | x <- [1..4], y <- [1..4], y > x]
[[1,2,3,4],[1,3,4],[1,4],[2,3,4],[2,4],[3,4]]
```

Bibliografía I


R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
Capítulo 4: Listas


G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
Chapter 5: List comprehensions


B. O'Sullivan, D. Stewart y J. Goerzen. *Real World Haskell*. O'Reilly, 2008.
Chapter 12: Barcode Recognition


B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. Razonando con Haskell. Thompson, 2004..
Capítulo 6: Programación con listas


S. Thompson. Haskell: The Craft of Functional Programming, Second Edition. Addison-Wesley, 1999.
Chapter 5: Data types: tuples and lists