PROGRAMANDO EN HASKELL

Capítulo 5 - Listas Comprendidas

Notación de Conjuntos

En Matemáticas, la notación de conjuntos puede usarse para construir nuevos cojuntos de conjuntos existentes.

$\{x^2 \mid x \in \{1...5\}\}$

El conjunto {1,4,9,16,25} de todos los números x^2 tal que x es un elemento del conjunto {1…5}.

Listas Comprendidas

En Haskell, una notación similar puede usarse para construir nuevas listas de listas ya existentes.

```
[x^2 | x ← [1..5]]
```

La lista [1,4,9,16,25] de todos los números x^2 tal que x es un elemento de la lista [1..5].

Nota:

- La expresión x ← [1..5] se llama generador, y especifica como generar valores para x.

- Las listas comprendidas pueden tener múltiples generadores, separados por comas. Ejemplo:

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

[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

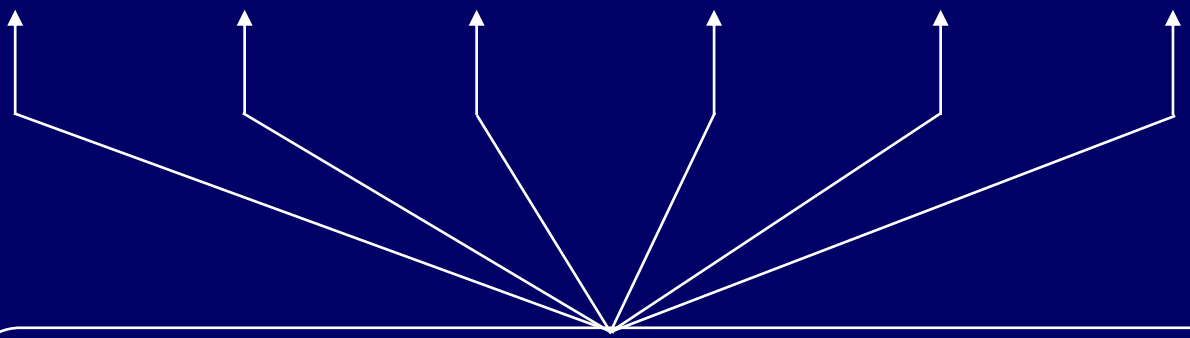
- Cambiar el orden de los generadores cambia el orden de los elementos en la lista final:

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

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

- Los generadores múltiples son como ciclos anidados, siendo los generadores posteriores como ciclos mas anidados cuyas variables cambian con mas frecuencia.

- Por ejemplo:

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

[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```



x ← [1,2,3] es el último generador, así es que el valor del componente x de cada par cambia con mas frecuencia.

Dependencia en Generadores

Los generadores posteriores pueden depender de las variables que han introducido los primeros generadores.

```
[(x,y) | x ← [1..3], y ← [x..3]]
```

La lista [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)] de todos los pares de números (x,y) tal que x,y son elementos de la lista [1..3] con $y \geq x$.

Usando dependencia en un generador podemos definir la función de librería que concatena una lista de listas:

```
concat :: [[a]] → [a]
concat xss = [x | xs ← xss, x ← xs]
```

Por ejemplo:

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

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

Guardas

Las listas comprendidas pueden usar guardas para restringir los valores producidos por los generadores que se especifican primero.

```
[x | x ← [1..10], even x]
```

La lista [2,4,6,8,10] de todos los números x tal que x es un elemento de la lista [1..10] siendo x un número par.

Con un guarda podemos definir una función que mapea un entero positivo a su lista de factores:

```
factors :: Int → [Int]
factors n =
   [x | x ← [1..n], n `mod` x == 0]
```

Por ejemplo:

```
> factors 15

[1,3,5,15]
```

Un entero positivo n es primo si sus únicos factores son 1 y n. Así, usando factors definimos una función que decide si un número es primo:

```
prime :: Int → Bool
prime n = factors n == [1,n]
```

Por ejemplo:

```
> prime 15
False

> prime 7
True
```

Con un guarda ahora podemos definir una función que regresa la lista de todos los primos hasta un límite establecido:

```
primes :: Int → [Int]
primes n = [x | x ← [2..n], prime x]
```

Por ejemplo:

```
> primes 40

[2,3,5,7,11,13,17,19,23,29,31,37]
```

La Función Zip

Una función de librería útil es zip, la cual mapea dos listas a la lista de pares de sus elementos correspondientes.

```
zip :: [a] → [b] → [(a,b)]
```

Por ejemplo:

```
> zip ['a','b','c'] [1,2,3,4]

[('a',1),('b',2),('c',3)]
```

Podemos definir una función que regresa la lista de todos los pares de elementos adyacentes en una lista, usando zip:

```
pairs :: [a] → [(a,a)]
pairs xs = zip xs (tail xs)
```

Por ejemplo:

```
> pairs [1,2,3,4]

[(1,2),(2,3),(3,4)]
```

Usando pairs Podemos definir una función que nos dice si los elementos de una lista están ordenados:

```
sorted :: Ord a ⇒ [a] → Bool
sorted xs = and [x ≤ y | (x,y) ← pairs xs]
```

Por ejemplo:

```
> sorted [1,2,3,4]
True

> sorted [1,3,2,4]
False
```

Usando zip podemos definir una función que regresa la lista de todas las posiciones de un valor en una lista:

```
positions :: Eq a ⇒ a → [a] → [Int]
positions x xs =
   [i | (x',i) ← zip xs [0..], x == x']
```

Por ejemplo:

```
> positions 0 [1,0,0,1,0,1,1,0]
[1,2,4,7]
```

Cadenas Comprendidas

Una cadena es una secuencia de caracteres entrecomillados. Sin embargo, internamente, las cadenas se representan como listas de caracteres.

```
"abc" :: String
```

Significa ['a', 'b', 'c'] :: [Char].

Debido a que las cadenas son solo una clase especial de listas, cualquier función polimórfica que opera sobre listas también puede aplicarse a cadenas. Por ejemplo:

```
> length "abcde"
5

> take 3 "abcde"
"abc"

> zip "abc" [1,2,3,4]
[('a',1),('b',2),('c',3)]
```

De manera similar, las listas comprendidas pueden usarse para definir funciones sobre cadenas, como contar cuantas veces un carácter aparece en una cadena:

```
count :: Char → String → Int
count x xs = length [x' | x' ← xs, x == x']
```

Por ejemplo:

```
> count 's' "Mississippi"
4
```

Ejercicios

(1) Un trío (x,y,z) de enteros positivos es pitagórico si $x^2 + y^2 = z^2$. Usando listas comprendidas, define una función

```
pyths :: Int → [(Int,Int,Int)]
```

que mapea un entero n a todos los tríos mencionados con componentes en [1..n]. Por ejemplo:

```
> pyths 5
[(3,4,5),(4,3,5)]
```

(2) Un entero positivo es perfecto si es igual a la suma de todos sus factores, excluyendo el número en sí mismo. Con una lista comprendida, define una función

```
perfects :: Int → [Int]
```

que regresa la lista de todos los números perfectos hasta el límite establecido. Por ejemplo:

```
> perfects 500

[6,28,496]
```

(3) El producto escalar de dos listas de enteros xs y ys de tamaño n está dado por la suma de los productos de los enteros correspondientes:

$$\sum_{i=0}^{n-1} (xs_i * ys_i)$$

Usando una lista comprendida, define una función que regresa el producto escalar de dos listas.

Fuente.

Profesor Graham Hutton.
http://www.cs.nott.ac.uk/~pszgmh/

Traducción al español.

Caleb Josue Ruiz Torres.
https://www.calebjosue.com