

Prácticas de laboratorio de LTP (Parte II : Programación Funcional)

Práctica 5: Listas y tipos algebraicos (I)



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA

Jose Luis Pérez
jlperez@dsic.upv.es

Introducción

En esta nueva práctica, que nos ocupará 2 sesiones de laboratorio seguiremos profundizando en los elementos más importantes del lenguaje Haskell, tales como, **listas** y los **tipos algebraicos** :

- La **lista** es la estructura de datos más utilizada y pueden ser utilizadas de diferentes formas para modelar y resolver muchos tipos de problemas.
- También estudiaremos las funciones **map** y **filter** como introducción al estudio del **orden superior**.
- En la siguiente sesión veremos como el programador puede definir nuevos tipos, en el lenguaje Haskell, con sus valores asociados, empleando los llamados **tipos algebraicos**.

Nota: En Poliformat se dispone de un enlace a un libro de Haskell en castellano. Y también el fichero **codigoEnPdf_P5** que podéis utilizar para copiar y pegar los ejemplos que se presentan durante la sesión.

El tipo Lista

En programación funcional es posible emplear tipos estructurados cuyos valores están compuestos por objetos de otros tipos.

En Haskell, las listas pueden especificarse encerrando sus elementos entre corchetes y separándolos con comas:

```
a :: [Int]
```

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

a es una lista de enteros (tipo **Int**)

```
b :: [Char]
```

```
b = ['a','b','c','d']
```

b es una lista de caracteres (tipo **Char**)

```
c :: [Float -> Float]
```

```
c = [cos, log, (1.0 +)]
```

c es una lista de funciones unitarias (tipo **Float -> Float**)

Si intentamos introducir en GHCi cualquiera de estas expresiones se producirá un **error**, ya que Haskell no es capaz de encontrar un tipo común a todos los elementos:

```
[1, 'a', 2]  
['a', log, 3]  
[cos, 2, (*)]
```

Funciones polimórficas sobre listas

Los tipos que incluyen variables de tipo en su definición (como el tipo lista) son tipos genéricos o polimórficos. Se pueden definir funciones sobre tipos polimórficos, que pueden emplearse sobre objetos de cualquier tipo que sea una instancia de los tipos polimórficos involucrados.

Por ejemplo, la función (predefinida) **length** calcula la longitud de una lista:

```
Prelude> :t length  
length :: [a] -> Int
```

a representa un tipo genérico (está en minúscula!!).

```
Prelude> length [1,2,3]  
3  
Prelude> length ['a','b','c','d']  
4  
Prelude> length [cos,log,sin]  
3
```

El operador (**!!**) permite la indexación de listas, y puede usarse con listas de cualquier tipo base:

```
Prelude> :t (!!)  
(!!) :: [a] -> Int -> a
```

```
Prelude> [1,2,3] !! 2  
3  
Prelude> ['a','b','c','d'] !! 0  
'a'
```

El operador (**++**) permite concatenar dos listas de cualquier tipo:

```
Prelude> :t (++)  
(++) :: [a] -> [a] -> [a]
```

```
Prelude> [1,2,3] ++ [4,5,6]  
[1,2,3,4,5,6]  
Prelude> ['a','b','c','d'] ++ ['e','f']  
"abcdef"
```

Operador :

La lista vacía se denota como []. Cuando no es vacía, se puede descomponer usando una notación que separa el elemento inicial (*cabeza*), de la lista que contiene los elementos restantes (*cola*):

cabeza es la posición 0 de la nueva **lista** creada

cabeza : *cola*

cola es otra **lista**

Por ejemplo:

Cualquiera de estas expresiones equivale a [1,2,3]

1:[2,3] o 1:2:[3] o 1:2:3:[]

'a':['b','c','d'] o 'a':'b':['c','d'] o 'a':'b':'c':'d':[]

cos:[log] o cos:log:[]

Cualquiera de estas expresiones equivale a ['a','b','c','d']. Y además las cadenas de texto tienen una notación especial que permite expresarlas como "abcd"

El operador : que se utiliza para construir una nueva lista añadiendo un elemento (*cabeza*) a una lista (*cola* de la nueva lista resultado) se puede utilizar también para deconstruir una lista. Veamos un ejemplo...

```
Prelude> a:b = [1,2,3,4]
```

```
Prelude> a  
1
```

```
Prelude> b  
[2,3,4]
```

La variable **a** contiene el elemento *cabeza* de la lista

La variable **b** contiene una nueva lista compuesta por la *cola* de la lista original

Los rangos (listas aritméticas)

La forma básica de las listas aritméticas o rangos tiene la sintaxis:

[first..last]

de modo que genera la lista de valores entre ambos (inclusive):

```
Prelude> [10..15]  
[10,11,12,13,14,15]
```

Si solo se expresa **first** y **last** se considera que el incremento es **1**

La sintaxis de los rangos en Haskell permite las siguientes opciones:

[first..] **[first,second..]** **[first..last]** **[first,second..last]**

Cuyo comportamiento se puede deducir de los siguientes ejemplos:

```
[0..]      -> 0, 1, 2, 3, 4, ...  
[0,10..]   -> 0, 10, 20, 30, ...  
[0,10..50] -> 0, 10, 20, 30, 40, 50  
[10,10..]  -> 10, 10, 10, 10, ...  
[10,9..1]  -> 10, 9, 8, 7, 6, 5, 4, 3, 2, 1  
[1,0..]    -> 1, 0, -1, -2, -3, ...  
[2,0..(-10)] -> 2, 0, -2, -4, -6, -8, -10
```

Haskell permite crear listas que representan conjuntos con un número infinito de elementos

En estos ejemplos el incremento (positivo o negativo) es determinado por **second**

Estrategia de reducción *Lazy* (perezosa)

La estrategia de reducción en Haskell es *lazy* (perezosa). Gracias a esto, es posible trabajar con estructuras de datos infinitas.

La función de Haskell **repeat** es también una función genérica, que devuelve una lista infinita, con el mismo elemento repetido:

```
Prelude> :t repeat
repeat :: a -> [a]
```

```
Prelude> repeat 3
[3,3,3,3,3,3,3,3,3,3,...
```

Una lista infinita generada por **repeat** puede ser usada como argumento parcial por una función que tiene un resultado finito. La función de Haskell, **take**, por ejemplo, toma un número finito de elementos de una lista:

```
Prelude> :t take
take :: -> Int -> [a] -> [a]
```

```
Prelude> take 4 (repeat 3)
[3,3,3,3]
```

Veamos una posible implementación alternativa para la función **take**:

```
take' :: Int -> [a] -> [a]
take' [] = []
take' n (x:t)
  | n <= 0 = []
  | otherwise = x : take' (n-1) t
```

En el primer patrón determina que si la cadena es vacía, en tal caso el resultado también lo es

En el caso general se utiliza un patrón que separa la cabeza de la cola de la lista, con el fin de aplicar la recursividad sobre la cola.

Listas intensionales

Haskell proporciona una notación alternativa para las listas, las llamadas **listas intensionales**, que permite la definición de conjuntos de elementos de una manera muy potente:

He aquí un ejemplo:

```
Prelude> [x * x | x <- [1..5], odd x]  
[1,9,25]
```

La expresión se lee: “la lista de los cuadrados de los números impares en el rango de 1 a 5”

Formalmente, una lista intensional se expresa de la forma:

[expresión | generador_o_restriccion, ..., generador_o_restriccion]

- ✓ Donde un **generador** toma la forma: $x \leftarrow xs$ { x es una variable o tupla de variables
- ✓ Y una **restricción** es una expresión **booleana** } xs es una expresión de tipo lista

He aquí algunos ejemplos:

```
Prelude> [(a,b) | a <- [1..3], b <- [1..2]]  
[(1,1),(1,2),(2,1),(2,2),(3,1),(3,2)]
```

La expresión es una tupla, y no hay restricción

```
Prelude> [(a,b) | a <- [1..3], b <- [1..2], a/=b]  
[(1,2),(2,1),(3,1),(3,2)]
```

En este caso hay restricción, no se permite que los elementos de la tupla sean iguales

```
Prelude> [(a,b) | a <- [1..2], b <- [1..3]]  
[(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

El orden de los generadores puede ser relevante

1. Las listas : Ejercicios resueltos

Ejercicio 1: Definir una función para calcular el valor binario correspondiente a un número entero no negativo con el siguiente perfil:

```
decBin :: Int -> [Int]
```

```
> decBin 4  
[0,0,1]
```

El último dígito es el más significativo

```
decBin :: Int -> [Int]
```

```
decBin 0 = [0]
```

```
decBin 1 = [1]
```

```
decBin x = (mod x 2):decBin (div x 2)
```

Se calcula el bit menos significativo y se añade a la cabeza de la lista

Utilizamos ajuste de patrones para especificar los casos base de la recursión (La solución aquí propuesta es diferente a la proporcionada en la documentación de la práctica)

Ejercicio 2: Definir una función para calcular el valor decimal correspondiente a un número en binario (representado como una lista de 1's y 0's):

```
binDec :: [Int] -> Int
```

```
> binDec [0,1,1]  
6
```

En el caso base se utiliza un patrón que representa una lista con un solo elemento (el uso de paréntesis es obligatorio)

```
binDec :: [Int] -> Int
```

```
binDec (x:[]) = x
```

```
binDec (x:y) = x + binDec y * 2
```

1. Las listas : Ejercicios propuestos (I)

Ejercicio 3: Definir una función para calcular la lista de divisores de un número entero no negativo:

```
divisors :: Int -> [Int]
```

```
> divisors 24  
[1,2,3,4,6,8,12,24].
```

Solución

```
divisors :: Int -> [Int]  
divisors n = [x | x <- [1..n] , (n `mod` x) == 0]
```

Ejercicio 4: Definir una función para determinar si un entero pertenece a una lista de enteros:

```
member :: Int -> [Int] -> Bool
```

```
> member 1 [1,2,3,4,8,9]  
True  
> member 0 [1,2,3,4,8,9]  
False
```

Ejercicio 5: Definir una función para comprobar si un número es primo (sus divisores son 1 y el propio número) y una función para calcular la lista de los n primeros números primos:

```
isPrime :: Int -> Bool
```

```
primes :: Int -> [Int]
```

```
> isPrime 2  
True  
> primes 5  
[2,3,5,7]
```

1. Las listas : Ejercicios propuestos (II)

Ejercicio 6: Definir una función para seleccionar los elementos pares de una lista de enteros:

```
selectEven :: [Int] -> [Int]
```

```
> selectEven [1,2,4,5,8,9,10]  
[2,4,8,10]
```

Ejercicio 7: Definir ahora una función para seleccionar los elementos que ocupan las “posiciones pares” de una lista de enteros (recuerda que las posiciones en una lista empiezan por el índice cero, siguiendo el funcionamiento del operador !!):

```
selectEP :: [Int] -> [Int]
```

```
> selectEP [1,2,4,5,8,9,10]  
[1,4,8,10]
```

Ejercicio 8: Definir una función **iSort** para ordenar una lista en sentido ascendente. Para ello, definir antes una función **ins** que inserte correctamente un elemento en una lista ordenada (la ordenación se puede resolver recursivamente, considerando sucesivas operaciones de inserción de los elementos a ordenar en la parte de la lista ya ordenada):

```
ins :: Int -> [Int] -> [Int]
```

```
iSort :: [Int] -> [Int]
```

```
> ins 5 [0,3,4,6,9]  
[0,3,4,5,6,9]
```

```
> iSort [4,9,1,3,6,8,7,0]  
[0,1,3,4,6,7,8,9]
```

2. Las funciones *map* y *filter*

Se trata de dos funciones predefinidas útiles para operar con listas. Son funciones muy comunes de las denominadas de orden superior al aceptar como argumentos (en este caso el primero de ellos) funciones.

La función **map** aplica una función a cada elemento de una lista:

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

```
> map square [9,3]
[81,9]
> map (<3) [1,2,3]
[True,True,False]
```

La función **filter** toma una función booleana **p** y una lista **xs** y devuelve la sublista de **xs** cuyos elementos satisfacen **p**.

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

```
> filter even [1,2,4,5,32]
[2,4,32]
```

2. Las funciones *map* y *filter* : Ejercicios propuestos

Ejercicio 9: Definir, usando la función `map`, una función para duplicar todos los elementos de una lista de enteros:

```
doubleAll :: [Int] -> [Int]
```

```
> doubleAll [1,2,4,5]  
[2,4,8,10]
```

Ejercicio 10: Expresar mediante listas intensionales las definiciones de las funciones **map** y **filter**. **Nota:** Se las puede llamar **map'** y **filter'** para evitar conflictos con las funciones predefinidas del **Prelude**.