Programación con Listas

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

1542

Escuela de Ingeniería y Arquitectura

Universidad Zaragoza

Departamento de Informática e Ingeniería de Sistemas

Universidad Zaragoza

1542

2023-01-20

Objetivos

Objetivos

El objetivo de este tema es:

- Conocer las diferentes formas de construir listas (listas aritméticas, generadores...)
- Conocer las diferentes funciones para manipular listas.
- Estudiar las funciones de orden superior.

```
1 l  = [1,2,3,4,5,6,7,8,9] :: [Int]
```

Constructores:

- `[]` construye una lista vacía
- `(:)` construye una lista a partir de un elemento y otra lista

Funciones básicas:

- Concatenación: `[2,3] ++ [4,5]`
- Acceso: `head tail las init take drop (!!)`
- Otros: length

Orden superior: `map`, `filter` ...

Construcción

Generación

Hay dos constructores básicos para listas:

- `[]` lista vacía
- `(:)` añade un elemento al principio de una lista

A partir de ellos se puede construir cualquier otra lista:

```
1:[]
1:(2:(3:[]))
```

...pero la notación es farragosa y existen notaciones simplificadas:

```
1:2:3:[]  -- el constructor : es asociativo por la derecha
[1,2,3]   -- notacion simplificada
```

Simple:

```
1 ghci> [1,2,3] ++ [7,8,9]
2 [1,2,3,7,8,9]
```

Múltiple:

```
1 ghci> concat [ [1,2,3], [4,5], [6,7,8,9] ]
2 [1,2,3,4,5,6,7,8,9]
```

Existe una sintaxis compacta para la defición de listas aritméticas:

```
1  ghci> [1..9]
2  [1,2,3,4,5,6,7,8,9]
3
4  ghci> [1,3..9]
5  [1,3,5,7,9]
6
7  ghci> ['a'..'z']
8  "abcdefghijklmnopqrstuvwxyz"
9
10 ghci> ['a','c'..'z']
11 "acegikmoqsuwy"
```

```
1  ghci> [9..1]
2  []
3
4  ghci> [9,8..1]
5  [9,8,7,6,5,4,3,2,1]
6
7  ghci> [0.0, 0.5 .. 3]
8  [0.0,0.5,1.0,1.5,2.0,2.5,3.0]
9
10 ghci> [3.0,2.5 .. 0.0]
11 [3.0,2.5,2.0,1.5,1.0,0.5,0.0]
```

¿Qué hace esta función?

```
1    cuenta :: Int -> Int -> [Int]
2    cuenta n m = if n <= m
3                 then n:(cuenta (n+1) m)
4                 else []
```

¿Qué hace esta función?

```
1   cuenta :: Int -> Int -> [Int]
2   cuenta n m = if n <= m
3                then n:(cuenta (n+1) m)
4                else []
```

La defición de listas mediante secuencias aritméticas no es más que una forma corta de escribir una función.

¿Y esta?

```
cuenta :: Int -> [Int]
cuenta n = n : cuenta (n+1)
```

¿Y esta?

```
1   cuenta :: Int -> [Int]
2   cuenta n = n : cuenta (n+1)
```

En Haskell es posible definir listas infinitas:

```
1   a = [1..]
```

Qué pasa en estos dos casos ?

```
1   ghci> print [1..]
2   ghci> print (head [1..])
```

Evaluación perezosa: recordad que Haskell no **evalúa** los argumentos antes de llamar a una función.

Se **sustituye** la definición y sus argumentos hasta que pueda decidirse el resultado.

```
1 cuenta :: Int -> [Int]
2 cuenta n = n : cuenta (n+1)
3
4 head :: [a] -> a
5 head (x:_) = x
```

```
1 ghci> head (cuenta 1)
2 -- definicion de cuenta
3 head (1:(cuenta 2))
4 -- definicion de head
5 1
```

Esto permite manejar listas infinitas evaluando sólo la parte que se necesita.

Evaluación perezosa: recordad que Haskell no **evalúa** los argumentos antes de llamar a una función.

Se **sustituye** la definición y sus argumentos hasta que pueda decidirse el resultado.

```haskell
head :: [a] -> a
head (x:_) = x
```

```haskell
ghci> head [1..]
-- definicion de [..]
head 1:[2..]
1
```

Esto permite manejar listas infinitas evaluando sólo la parte que se necesita.

En Matemáticas existe la forma de definir conjuntos denominada **por comprensión** (frente a la definición por enumeración):

$$[\, x \mid x \in \mathbb{N},\ x \textit{ es par} \,]$$

En Haskell podemos hacer algo similar para definir listas.
Nuestro conjunto de partida puede ser una lista finita o infinita:

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

La sintaxis general es:

$$[\,\textit{expresion} \mid q_1 , q_2 , q_3, \ldots\,]$$

donde cada **cualificador** q_i puede ser:

- un **generador**, expresión que genera una lista: `x <- [0..10]`
- una **guarda**, o expresión booleana: even x
- una **definición** local: let sq = x*x

Y la **expresión** puede evaluarse usando las definiciones creadas en los cualificadores:

```
1  [ 2*sq+x+1 | x<-[1..10], even x, let sq = x*x ]
```

Los generadores pueden definir variables...
...que las guardas pueden filtrar...
...usando definiciones locales para hacer el código más legible...
...y esos valores se utilizan para evaluar la expresión:

```
1  ghci> [ 2*x | x <- [1..5], even x ]
2  [4,8]
3
4  ghci> [ 2*sq+x+1 | x<-[1..10], even x, let sq = x*x ]
5  [11,37,79,137,211]
```

Puede haber varias guardas:

```
1   [ x | x <- [1..20], even x, mod x 3 == 0 ]
2   [6,12,18]
```

Y varios generadores (el de más a la derecha varía más rápido):

```
1   [ (x,y) | x <- [1..5], even x, y <- ['a'..'c'] ]
2   [(2,'a'),(2,'b'),(2,'c'),(4,'a'),(4,'b'),(4,'c')]
```

Ofrecen un mecanismo equivalente a los bucles anidados:

```
1   [(i,j) | i<-[1..2], j<-[1..3]]
2   [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

¿Qué hacen estas funciones?

```
1    f :: Int -> [Int]
2    f n = [ k | k <- [1..n], mod n k == 0 ]
3
4    g :: Int -> Int-> [Int]
5    g n m = [ k | k <- [1..min n m],
6                   mod n k == 0, mod m k == 0 ]
7
8    h :: Int -> Int-> Int
9    h n m = maximum [ k | k <- [1..min n m],
10                          mod n k == 0, mod m k == 0 ]
```

Ejemplo:

```
1  divisores :: Int -> [Int]
2  divisores n = [ d | d <- [1..n], mod n d == 0 ]
3
4  prime :: Int -> Bool
5  prime n = (divisores n) == [1,n]
6
7  primes :: Int -> [Int]
8  primes n = [ x | x <- [1..n], prime x ]
```

Podemos definir variables locales. Son visibles en el resto de los cualificadores y en la expresión.

Ejemplo:

```
1  [ cuadrado | x <- [1..10],
2               let cuadrado = x*x, even cuadrado ]
3
4  [4,16,36,64,100]
```

Acceso

Acceso estándar:

```
ghci> head [1, 2, 3]
1
ghci> tail [1, 2, 3]
[2,3]
```

Acceso inverso:

```
ghci> init [1, 2, 3]
[1,2]
ghci> last [1, 2, 3]
3
```

Acceso indexado:

```
1  ghci> [1, 2, 3, 4, 5] !! 0
2  1
3  ghci> [1, 2, 3, 4, 5] !! 2
4  3
5  ghci> [1, 2, 3, 4, 5] !! 4
6  5
7  ghci> [1, 2, 3, 4, 5] !! 6
8  *** Exception: Prelude.(!!): index too large
9  ghci> [1, 2, 3, 4, 5] !! (-1)
10 *** Exception: Prelude.(!!): negative index
```

Podemos extraer segmentos de listas (*slices*):

```
1   ghci> take 2 [1..5]
2   [1,2]
3
4   ghci> drop 2 [1..5]
5   [3,4,5]
```

```
1   ghci> splitAt 2 [1 .. 5]
2   ([1,2],[3,4,5])
```

Especialmente útil para trabajar con listas infinitas:

```
1   ghci> drop 3 (take 5 [20..])
2   [23,24]
3   ghci> take 5 (drop 3 [20..])
4   [23,24,25,26,27]
```

```
1   ghci> (a,b) = splitAt 10 [1..]
2   ghci> a
3   [1,2,3,4,5,6,7,8,9,10]
4   ghci> head b
5   11
```

zip empareja dos listas generando parejas (tuplas) de elementos:

```
1  ghci> zip [1,2] ['a','b']
2  [(1,'a'),(2,'b')]
```

Las listas no tienen por que ser del mismo tamaño:

```
1  ghci> zip [1,2,3,4] ['a','b']
2  [(1,'a'),(2,'b')]
```

Y en particular...

```
1  ghci> zip [1..] ['a','b']
2  [(1,'a'),(2,'b')]
```

La función **unzip** realiza el proceso inverso.

```
1  ghci> unzip [(1,'a'),(2,'b'),(3,'c'),(4,'d')]
2  ([1,2,3,4],"abcd")
```

Funciones de Orden Superior

La función **map** aplica una función a todos los elementos de una lista:

```
1  ghci> sqr x = x*x
2  ghci> map sqr [1..5]
3  [1,4,9,16,25]
4
5  ghci> map odd [1..5]
6  [True,False,True,False,True]
7
8  ghci> map (2 *) [1..5]
9  [2,4,6,8,10]
```

La función **zipWith** empareja dos listas aplicando una función a los dos argumentos, tomando uno de cada lista:

```
1  ghci> add x y = x+y
2  ghci> zipWith add [1..5] [10..]
3  [11,13,15,17,19]
4  ghci> zipWith (+) [1,2] [3,4]
5  [4,6]
```

En particular...

```
1  ghci> zipWith (,) [1..] "abcd"
2  [(1,'a'),(2,'b'),(3,'c'),(4,'d')]
```

Un filtro nos permite seleccionar los elementos de una lista que cumplen una determinada propiedad (**predicado**).

En Haskell podemos implementar filtros mediante la función **filter**:

```
1  filter :: (a -> Bool) -> [a] -> [a]
2
3  ghci> filter even [1..10]
4  [2,4,6,8,10]
5
6  ghci> filter (> 'g') "me gustan las listas"
7  "mustnlslists"
```

Hola

Otros filtros:

- `takeWhile`: devuelve elementos mientras se cumpla el predicado

```
1  takeWhile even [2,4,6,8,9,10,11,13]
2  [2,4,6,8]
```

- `dropWhile`: descarta elementos mientras se cumpla el predicado

```
1  dropWhile even [2,4,6,8,9,10,11,13]
2  [9,10,11,13]
```

Y en particular... ¿qué hace este código ?

```
1 takeWhile (< 100) (map (^2) [0..])
2 [0,1,4,9,16,25,36,49,64,81]
```

¿Y por qué funciona con una lista infinita?

Ejercicio

Programa en Haskell tu propia versión de la función **takeWhile**

Las operaciones de reducción se implementan en Haskell mediante la función de plegado o **fold**:

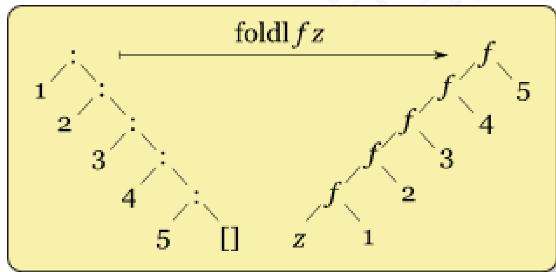
```
1  fold op acc list -> acc
```

de la que existen dos versiones: **foldr** y **foldl**:

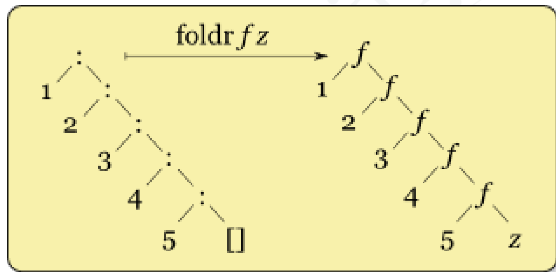
```
1  ghci> foldr (+) 0 [1..5]
2  15
3  ghci> foldl (+) 0 [1..5]
4  15
```

¿Cuál es entonces la diferencia?

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



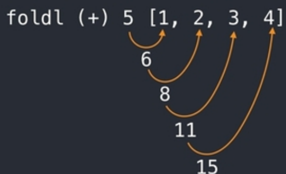
```haskell
foldr :: (a->b->b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

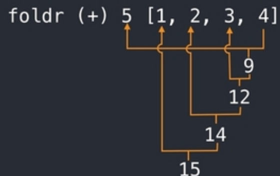


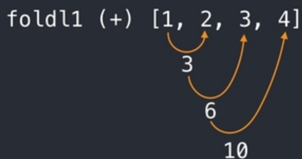
Ejemplo:

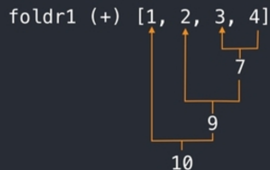
$$1 + 2 + 3 + 4 + 5$$

```
1 foldl (+) 0 [1..5]
2     ((((0 + 1) + 2) + 3) + 4) + 5
3
4 foldr (+) 0 [1..5]
5     1 + (2 + (3 + (4 + (5 + 0))))
```


foldl
foldl (+) 5 [1, 2, 3, 4]
6
8
11
15


foldr
foldr (+) 5 [1, 2, 3, 4]
9
12
14
15


foldl1
foldl1 (+) [1, 2, 3, 4]
3
6
10


foldr1
foldr1 (+) [1, 2, 3, 4]
7
9
10

La existencia de dos funciones tiene sentido con operaciones binarias *no conmutativas*

```
foldl (**) 2 [1, 2, 3]
    (((2 ** 1) ** 2) ** 3) = 64

foldr (**) 2 [1, 2, 3]
    (1 ** (2 ** (3 ** 2))) = 1
```

Para operaciones sencillas puede verse el **fold** como una *reducción* que utiliza un acumulador:

```
foldl :: (b->a->b) -> b -> [a] -> b
foldl op acc0 [x1,x2,x3,x4....xn]
  acc <> [x1,x2,x3,x4....xn]
  acc = acc0
  acc = op acc x1 -- = acc `op` x1
  acc = op acc x2 -- = acc `op` x2
  ...
  acc = op acc xn -- = acc `op` xn
```

Para operaciones sencillas puede verse el **fold** como una *reducción* que utiliza un acumulador:

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr op acc0 [x1,x2,x3,x4....xn]
  [x1,x2,x3,x4....xn] <> acc
  acc = acc0
  acc = op xn acc -- = xn `op` acc
  ...
  acc = op x2 acc -- = x2 `op` acc
  acc = op x1 acc -- = x1 `op` acc
```

No hay que olvidar que el **currying** permite **aplicación parcial**:

- La función map no sólo transforma, genera transformaciones.

```
1 ghci> cesar d = map (chr.(+(ord 'a')).(`mod` 26).(+ d).(+ (- ord 'a')).ord)
2 ghci> cesar 7 "haskell"
3 "ohzrlss"
```

- La función `filter` no sólo filtra, genera filtros.

```
1 ghci> quitapares = filter odd
2 ghci> quitapares [1..9]
3 [1, 3, 5, 7, 9]
```

```haskell
cesar d = map (
  chr
  .(\n -> n + (ord 'a'))
  .(\n -> n mod 26)
  .(\n -> n + d)
  .(\n -> n - (ord 'a'))
  .ord
)
```

```haskell
cesar d = map (
  chr
  .(+ (ord 'a'))
  .(`mod` 26)
  .(+ d)
  .(subtract (ord 'a'))
  .ord
)
```

La aplicación parcial se usa también para definir instrucciones estándar de plegado del `Prelude`

```
1  sum     = foldl (+) 0
2  product = foldl (*) 1
3  concat  = foldl (++) []
4
5  maximum = foldl1 max
6  minimum = foldl1 min
```

- `span`, `break`: combinación de thakeWhile y dropWhile
- `words`, `lines`: dividen un texto en palabras o lineas.

Ejemplo: ordenación por inserción

```haskell
-- Inserta un dato en una lista ya ordenada
insert :: Int -> [Int] -> [Int]
insert x []      = [x]
insert x l@(y:ys)
    | x <= y     = x : l
    | otherwise = y : insert x ys

-- Ordena una lista por el metodo de insercion
isort :: [Int] -> [Int]
-- Implementacion ??
```

```haskell
isort [7,4,5] =
insert 7 (isort [4,5])
insert 7 (insert 4 (isort [5]))
insert 7 (insert 4 (insert 5 []))
```

Luego...

```haskell
isort :: [Int] -> [Int]

isort l = foldr insert [] l
```

O mejor, con notación *point–free*:

```haskell
isort   = foldr insert []
```

```haskell
qsort :: [Int] -> [Int]

qsort []     = []
qsort [x]    = [x]
qsort (p:xs) = qsort menores ++ [p] ++ qsort mayores
               where
                 menores = [ x | x <- xs, x <  p]
                 mayores = [ x | x <- xs, x >= p]
```

Programación con Listas

Tecnología de Programación

Adolfo Muñoz – Juan Magallón

Grado en Ingeniería Informática

Universidad Zaragoza

1542

Escuela de Ingeniería y Arquitectura

Universidad Zaragoza

1542

Departamento de Informática e Ingeniería de Sistemas

Universidad Zaragoza

2023-01-20