Lección 02

Funciones de Alto Nivel

La función `iterate` es de tipo `(a -> a) -> a -> [a]` y ofrece el siguiente diálogo:

```
Prelude> take 10 (iterate (2*) 1)
[1,2,4,8,16,32,64,128,256,512]
Prelude> take 10 (iterate (\x -> (x+3)*2) 1)
[1,8,22,50,106,218,442,890,1786,3578]
Prelude>
```

La función `zip` es de tipo `[a] -> [b] -> [(a,b)]` y ofrece el siguiente diálogo

```
Prelude> zip [1,2,3] [9,8,7]
[(1,9),(2,8),(3,7)]
Prelude> zip [1,2,3] [9,8,7]
[(1,9),(2,8),(3,7)]
Prelude> zip [1,2,3,4,5] [9,8]
[(1,9),(2,8)]
Prelude> zip (take 5 (iterate (2*) 10)) (take 5 (iterate (2/) 10))
[(10,10.0),(20,0.2),(40,10.0),(80,0.2),(160,10.0)]
Prelude>
```

La función `unzip` es de tipo `[(a,b)] -> ([a],[b])` y ofrece el siguente diálogo:

```
Prelude> unzip [(1,2),(2,3),(3,4)]
([1,2,3],[2,3,4])
Prelude>
```

La función `zipWith` es de tipo `(a -> b -> c) -> [a] -> [b] -> [c]` y ofrece el siguente diálogo:

```
Prelude>  zipWith (+) [1,2,3] [3,2,1]
[4,4,4]
Prelude>
Prelude> zipWith (**) (replicate 10 5) [1..10]
[5.0,25.0,125.0,625.0,3125.0,15625.0,78125.0,390625.0,1953125.0,9765625.0]
Prelude> zipWith (\x y -> 2*x + y) [1..4] [5..8]
[7,10,13,16]
Prelude>
```

Observe que `zip` es un caso particular de `zipWith` en el siguiente código:

```
Prelude> zipWith (,) [1,2,3] [3,2,1]
[(1,3),(2,2),(3,1)]
```

Existen las funciones: `zip3`, `zip4`, `zip5`, `zip6`, `zip7`, `zipWith`, `zipWith3`, `zipWith4`, `zipWith5`, `zipWith6` y `zipWith7`.

La función `filter` es de tipo `(a -> Bool) -> [a] -> [a]` y devuelve una lista construida sobre los miembros de otra (su segundo argumento) que cumplen una condición dada en su primer argumento. Ofrece el siguiente diálogo:

```
Prelude> filter (>5) [1,2,3,4,5,6,7,8]
[6,7,8]
Prelude> filter odd [3,6,7,9,12,14]
[3,7,9]
Prelude> filter (\x -> length x > 4) ["aaaa","bbbbbbbbbbbb","cc"]
["bbbbbbbbbbbb"]
Prelude>
```

La función `map` es de tipo `(a -> b) -> [a] -> [b]` y devuelve una lista construida aplicando una función, su primer argumento, a todos las entradas de una lista pasada como segundo argumento. Ofrece el siguiente diálogo:

```
Prelude> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
Prelude> map (++ "!") ["Salta","ahora","el", "charco"]
["Salta!","ahora!","el!","charco!"]
Prelude> map (++ "!") ["Salta","ahora"]
["Salta!","ahora!"]
Prelude> map (replicate 3) [3..6]
[[3,3,3],[4,4,4],[5,5,5],[6,6,6]]
Prelude> map (map (^2)) [[1,2],[3,4,5,6],[7,8]]
[[1,4],[9,16,25,36],[49,64]]
Prelude> map fst [(1,2),(3,4),(7,8)]
[1,3,7]
```

La función `find` del módulo `Data.List` y con tipo `(a -> Bool) -> [a] -> Maybe a` devuelve el primer elemento de una lista, su segundo argumento, que satisface una condición o `Nothing` si no hay un tal elemento. La función `findIndex`, de tipo `(a -> Bool) -> [a] -> Maybe Int` devuelve el índice correspondiente. La función `FindIndices` de tipo `(a -> Bool) -> [a] -> [Int]` devuelve una lista con todos esos índices. Estas funciones ofrecen el siguiente diálogo:

```
Prelude> :m + Data.List
Prelude Data.List> find (>3) [0,2,4,6,8]
Just 4
Prelude Data.List> find (==3) [0,2,4,6,8]
Nothing
Prelude Data.List>  find even [2,4,6,8]
Just 2
Prelude Data.List> find (\x -> 5**x > 10000) [2,4,6,8]
Just 6.0
Prelude Data.List> findIndex (>3) [0,2,4,6,8]
Just 2
Prelude Data.List> findIndex (==3) [0,2,4,6,8]
Nothing
Prelude Data.List> findIndex even [2,4,6,8]
Just 0
Prelude Data.List> findIndex (\x -> 5**x > 10000) [2,4,6,8]
Just 2
Prelude Data.List> findIndices (>3) [0,2,4,6,8]
[2,3,4]
Prelude Data.List> findIndices (==3) [0,2,4,6,8]
```

```
[]
Prelude Data.List> findIndices even [2,4,6,8]
[0,1,2,3]
Prelude Data.List> findIndices (\x -> 5**x > 10000) [2,4,6,8]
[2,3]
Prelude Data.List>
```

La función `elemIndex` del módulo `Data.List` y con tipo `Eq => a -> [a] -> Maybe Int` el índice de la primera ocurrencia, si la hay, del valor en la lista. La función `elemIndices`, de tipo `Eq a => a -> [a] -> [Int]`, devuelve la lista de índices (en orden) de las entradas del segundo argumento que son iguales al primero. Ofrece el siguiente diálogo:

```
Prelude Data.List> elemIndex 2 [1,2,3,4,5]
Just 1
Prelude Data.List> elemIndex 2 [2,2,2,3,3,3,4,4,4]
Just 0
Prelude Data.List> elemIndex 10 [1,2,3,4,5]
Nothing
Prelude Data.List> elemIndex 'f' "abcdefghi"
Just 5
Prelude Data.List> elemIndices 3 [1,2,3,4]
[2]
Prelude Data.List> elemIndices 3 [1,3,2,3,3,4,3]
[1,3,4,6]
Prelude Data.List> elemIndices 'a' "abbacca"
[0,3,6]
Prelude Data.List>
```

La función `all`, de tipo `(a -> Bool) -> [a] -> Bool`, devuelve `True` si, y sólo si, cada uno de los items de la lista que es su segundo argumento cumplen la condición que es el primero. Ofrece el siguiente diálogo:

```
Prelude> all (<10) [1,3,5,7,9]
True
Prelude> all (==1) [1,1,0,1,1]
False
Prelude> all even [2,4,6,8,10]
True
Prelude> all (\x -> (x*x)/4 > 10) [5,10,15]
False
Prelude>
```

La función `any`, de tipo `(a -> Bool) -> [a] -> Bool`, devuelve `True` si, y sólo si, al menos uno de los items de la lista que es su segundo argumento cumplen la condición que es el primero. Ofrece el siguiente diálogo:

```
Prelude> any (1==) [0,1,2,3,4,5]
True
Prelude> any (>5) [0,1,2,3,4,5]
False
Prelude> any even [1,3,5,7,9]
False
```

```
Prelude> any (\x -> x*4>20) [1,2,3,4,5,6]
True
Prelude>
```

La función `foldl`, de tipo `(a -> b -> a) -> a -> [b] -> a`, toma el segundo argumento y el primer item de la lista y aplica la función a ellos, entonces alimenta a la función con este resultado y el segundo argumento y así sucesivamente. Ofrece el siguiente diálogo:

```
Prelude> :t foldl
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
Prelude> foldl (div) 64 [4,2,4]
2
Prelude> ((64 `div` 4) `div` 2) `div` 4
2
Prelude> foldl (div) 64 [8,2,4]
1
Prelude> ((64 `div` 8) `div` 2) `div` 4
1
Prelude> foldl max 5 [1,5,6]
6
Prelude> ((5 `max` 1) `max` 5) `max` 6
6
Prelude> foldl max 5 [1,2,3,4,5,6,7]
7
Prelude> foldl (div) 3 []
3
Prelude> foldl (\x y -> 2*x + y) 4 [1,2,3]
43
Prelude>
```

Aún podemos dar una bella aplicación de `foldl`, cual es considerar la siguiente función:

```
rev :: [a] -> [a]
rev = foldl (\ xs x -> x:xs) []
```

cuyo código, editado en `Prueba02.hs` ofrece el siguiente diálogo:

```
Prueba02> rev [0,1,2,3]
[3,2,1,0]
```

y que funciona como sigue:

```
*Prueba02> g = \ xs x -> x:xs
*Prueba02> ((([] `g` 0) `g` 1) `g` 2) `g` 3
[3,2,1,0]
*Prueba02>
```

La función `scanl`, de tipo `(a -> b -> a) -> a -> [b] -> [a]`, opera como `foldl` pero ofreciendo los resultados intermedios. Ofrece el siguiente diálogo:

```
Prelude> scanl (div) 64 [4,2,4]
[64,16,8,2]
Prelude> scanl (div) 3 []
[3]
Prelude> scanl max 5 [1,2,3,4]
[5,5,5,5,5]
Prelude> scanl max 5 [1,2,3,4,5,6,7]
[5,5,5,5,5,5,6,7]
Prelude> scanl (\x y -> 2*x + y) 4 [1,2,3]
[4,9,20,43]
Prelude>
```

La función `foldl1`, de tipo `(a -> a -> a) -> [a] -> a`, toma los primeros 2 items de la lista y aplica la función a ellos, seguidamente alimenta la función con este resultado y el tercer item y así sucesivamente. Ofrece el siguiente diálogo:

```
Prelude> foldl1 (+) [1,2,3,4]
10
Prelude> foldl1 (div) [64,4,2,8]
1
Prelude>  foldl1 (div) [12]
12
Prelude> foldl1 (&&) [1>2,3>2,5==5]
False
Prelude> foldl1 max [3,6,12,4,55,11]
55
Prelude> foldl1 (\x y -> (x+y) `div` 2) [3,5,10,5]
6
Prelude>
```

La función `scanl1`, de tipo `(a -> a -> a) -> [a] -> [a]`, opera como `foldl1` pero ofreciendo los resultados intermedios. Ofrece el siguiente diálogo:

```
Prelude> scanl1 (+) [1,2,3,4]
[1,3,6,10]
Prelude> scanl1 (div) [64,4,2,8]
[64,16,8,1]
Prelude> scanl1 (div) [12]
[12]
Prelude> scanl1 (&&) [3>1,3>2,4>6,5==5]
[True,True,False,False]
Prelude> scanl1 max [3,6,12,4,55,11]
[3,6,12,12,55,55]
Prelude> scanl1 (\x y -> (x+y)/2) [3,5,10,5]
[3.0,4.0,7.0,6.0]
Prelude>
```

La función `foldr`, de tipo `(a -> b -> b) -> b -> [a] -> b`, toma el segundo argumento y el último item de la lista y aplica la función, entonces toma el penúltimo item de la lista y el resultado y con ello alimenta de nuevo la función; así sucesivamente. Ofrece el siguiente diálogo:

```
Prelude> foldr (+) 5 [1,2,3,4]
15
Prelude> foldr (div) 2 [8,12,24,4]
8
*Prueba02> 8 `div` (12 `div` (24 `div` (4 `div` 2)))
8
Prelude> foldr (div) 3 []
3
Prelude> foldr (&&) True [1>2,3>2,5==5]
False
Prelude> foldr max 18 [3,6,12,4,55,11]
55
Prelude> foldr max 111 [3,6,12,4,55,11]
111
Prelude> foldr (\x y -> (x+y) `div` 2) 54 [12,4,10,6]
12
Prelude>
```

La función `scanr`, de tipo `(a -> a -> a) -> [a] -> [a]`, opera como `foldr` pero ofreciendo los resultados intermedios. Ofrece el siguiente diálogo:

```
Prelude> scanr (+) 5 [1,2,3,4]
[15,14,12,9,5]
Prelude> scanr (div) 2 [8,12,24,4]
[8,1,12,2,2]
Prelude> scanr (div) 3 []
[3]
Prelude> scanr (&&) True [1>2,3>2,5==5]
[False,True,True,True]
Prelude> scanr max 18 [3,6,12,4,55,11]
[55,55,55,55,55,18,18]
Prelude> scanr max 111 [3,6,12,4,55,11]
[111,111,111,111,111,111,111]
Prelude> scanr (\x y -> (x+y) `div` 2) 54 [12,4,10,6]
[12,12,20,30,54]
Prelude>
```

La función `floldr1`, de tipo `(a -> a -> a) -> [a] -> a`, toma los dos últimos items de la lista y les aplica la función, seguidamente toma el tercer elemento desde la cola y lo opera según la función con el resultado anterior, y así sucesivamente. Ofrece el siguiente diálogo:

```
Prelude> foldr1 (+) [1,2,3,4]
10
Prelude> foldr1 (div) [8,12,24,4]
4
Prelude> foldr1 (div) [12]
12
Prelude> foldr1 (&&) [1>2,3>2,5==5]
False
Prelude> foldr1 max [3,6,12,4,55,11]
55
```

```
Prelude> foldr1 (\x y -> (x+y) `div` 2) [12,4,10,6]
9
Prelude>
```

La función `scanr1`, de tipo `(a -> a -> a) -> [a] -> [a]`, opera como `foldr1` pero presenta los resultados intermedios. Ofrece el siguiente diálogo:

```
Prelude> scanr1 (+) [1,2,3,4]
[10,9,7,4]
Prelude> scanr1 (div) [8,12,24,2]
[8,1,12,2]
Prelude>  scanr1 (div) [12]
[12]
Prelude> scanr1 (&&) [1>2,3>2,5==5]
[False,True,True]
Prelude> scanr1 max [3,6,12,4,55,11]
[55,55,55,55,55,11]
Prelude> scanr1 (\x y -> (x+y) `div` 2) [12,4,10,6]
[9,6,8,6]
Prelude>
```

En general los resultados de aplicar `foldr` y `foldl` a unos mismos datos no coinciden:

```
Prelude> foldl (div) 2 [8,12,24,4]
0
Prelude> foldr (div) 2 [8,12,24,4]
8
Prelude> foldr (-) 2 [8,12,24,4]
18
Prelude> foldl (-) 2 [8,12,24,4]
-46
Prelude> foldr (-) 0 [8,12,24,4]
16
Prelude> foldl (-) 0 [8,12,24,4]
-48
```

Las expresiones `foldr f z xs` y `foldl f z xs` coinciden cuando, y sólo cuando, se cumplen las siguientes condiciones:

- `f` es asociativa como operación implícita (`(a f b) f c = a f (b f c)`).
- `xs` sea una instancia finita de un tipo plegable.

La función `flip`, de tipo `(a -> b -> c) -> b -> a -> c`, evalúa la función volteando los argumentos. Ofrece el siguiente diálogo:

```
Prelude> flip div 1 2
2
Prueba02> div 2 1
2
Prelude> flip (>) 3 5
True
Prueba02> 5 > 3
True
```

```
Prelude> flip mod 3 8
2
Prueba02> 8 `mod` 3
2
```

Ejercicios

1. Diseñe una función llamada `enumerationWithPattern` que tenga por argumento tres listas del tipo `[Int]` y que funcione así: por cada entrada de la primera lista (digamos `m`) tome su correpondiente de la segunda (digamos `n`) y si la correspondiente de la tercera es `r` que entonces genere la lista de enteros `[m,n..r]`, presentando todos estos resultados de acuerdo con la lista más corta de todos en una lista. Por ejemplo, debe ofrecer el siguiente diálogo:

   ```
   *EjerciciosPrac02> enumerationWithPattern   [1,2,3] [3,5,7,8] [5,10,15]
   [[1,3,5],[2,5,8],[3,7,11,15]]
   *EjerciciosPrac02> enumerationWithPattern   [] [3,5,7,8] [5,10,15]
   []
   *EjerciciosPrac02>
   ```

2. Diseñe una función llamada `multiplesOf` que dado un entero `n` y una lista de enteros `lst` produzca una lista con entradas las parejas `(True,m)` (resp. `(False,m)`) si `m` es múltiplo de `n` (resp. `m` no es múltiplo de `n`) y ello por cada `m` de `lst`. Por ejemplo, debe ofrecer el siguiente diálogo:

   ```
   *EjerciciosPrac02> multiplesOf  2 [1,3,4,5,8]
   [(False,1),(False,3),(True,4),(False,5),(True,8)]
   *EjerciciosPrac02> multiplesOf 0 [1,3,4,5,8]
   [(False,1),(False,3),(False,4),(False,5),(False,8)]
   *EjerciciosPrac02> multiplesOf 0 [1,3,4,5,0,8,0]
   [(False,1),(False,3),(False,4),(False,5),(True,0),(False,8),(True,0)]
   *EjerciciosPrac02> multiplesOf 0 []
   []
   *EjerciciosPrac02> multiplesOf 3 []
   []
   *EjerciciosPrac02>
   ```

3. Diseñe una función, de nombre digamos `oppositeInverse`, que dada una lista de números produzca a partir de ella otra cumpliendo que cada una de sus entradas sea el opuesto del inverso de la correspondiente entrada de la lista argumento, caso de no ser nula. Decida que hacer en el caso de una entrada nula en la lista argumento.

4. Diseñe una función, de nombre digamos `multiplesOfFind` de tipo `Int -> [Int] -> [Int]` que seleccione de la lista que es su segunto argumento los múltiplos del primero y ofrezca una lista con ellos.

5. Haciendo uso de `foldr` diseñe una función, digamos `longr`, que calcule la longitud de una lista.

6. Haciendo uso de `foldr` diseñe una función, digamos `revr`, que funciones sobre listas como `reverse`.

7. Haciendo uso de `foldl` diseñe una función, digamos `revl`, que funciones sobre listas como `reverse`.
8. Dé un ejemplo que ponga de manifiesto que para unos mismos argumentos `foldr` y `foldl` no dan el mismo resultado.
9. Diseñe una función eficiente de nombre `rotateLev` que gire hacia la izquierda un string no vacío el número de posiciones que se le indique. Por ejemplo, ofrecerá el siguiente diálogo:

```
*EjerciciosPrac02> rotateLev 3 "abcdefghijk"
"defghijkabc"
```

10. Diseñe una función eficiente de nombre `rotateDes` que gire hacia la derecha un string no vacío el número de posiciones que se le indique. Por ejemplo, ofrecerá el siguiente diálogo:

```
*EjerciciosPrac02> rotateDes 3 "abcdefghijk"
"ijkabcdefgh"
```