

Paradigmas de Lenguajes de Programación

Tomás Agustín Hernández



Programación Funcional

Consiste en definir funciones y aplicarlas para procesar información.
Las funciones son verdaderamente funciones (parciales):

- Aplicar una función no tiene efectos secundarios.
- A una misma entrada le corresponde siempre la misma salida.
- Las estructuras de datos son inmutables.

Las funciones, además son datos como cualquier otro:

- Se pueden pasar como parámetros.
- Se pueden devolver como resultados.
- Pueden formar parte de estructuras de datos. Ej.: Un árbol binario que en sus nodos hay funciones.

Expresiones

Son secuencias de símbolos que sirven para representar datos, funciones, y funciones aplicadas a los datos.
Una expresión puede ser:

- Un constructor: True, False, [], (:), 0, 1, 2.
 - Type Constructor: Es un constructor que se utiliza para crear un nuevo tipo.
 - Data Constructor: Se utiliza para crear valores de ese tipo.
 - Ej.: *data Color = Rojo | Verde | Azul*. Azul es un **data constructor** pues nos permite crear valores del tipo Color mientras que el Type Constructor es Color.
 - Ej.: *data Complejo = C Float Float*. C es una función que recibe dos Float y es un constructor de Complejo.
- Una variable: longitud, ordenar, x, xs (+), (*).
- La aplicación de una expresión a otra: ordenar lista, not True, (+) 1.

Función Parcialmente Aplicada

Una función parcialmente aplicada es una función a las cuales se llama otra función pero no se le proporcionan todos los argumentos.

Ahora, es importante que para que esta función parcialmente aplicada tenga sentido se le manden todos los parámetros.
Es una especie de $a \rightarrow b$ Ej.:

```
1 | add :: Int -> Int -> Int
2 | add x y = x + y
3 |
4 | add5 :: Int -> Int
5 | add5 = add 5
6 |
7 | add5 es una función que ejecuta la función parcial add pues le manda solo un parámetro y necesita dos.
```

Una buena pregunta entonces es ¿pero por qué es una función parcial add 5? Pues hay una función add5 que la implementa sin pasarle todos los parámetros directamente explícitamente.

```
1 | add5 3 -> Este 3 llega como "y" a add
2 |
3 | add5 = add 5
4 | add = 5 + y
5 | add = 5 + 3
6 | add = 8
```

Otro ejemplo

```
1 | const :: a -> b -> a
2 | const x y = x
3 |
4 | const (const 1) 2 -> const 1
5 | El llamado de (const 1) es una función parcialmente aplicada porque no envía el valor de y, sin embargo,
   | const (const 1) 2 no la aplica parcialmente porque le termina mandando los dos parámetros.
```

Aplicación de Expresiones

Es asociativa hacia la izquierda:

- $f\ x\ y \equiv (f\ x)\ y$
- $((((f\ a)\ b)\ c)\ d)$: Primero calcula el resultado que devuelve la expresión f enviando el valor de a . Nótese que la idea sería que $(f\ a)$ devuelva una expresión del tipo función pues luego le pasamos otro parámetro (b) .

Importancia de la Aplicación de las Expresiones

¿Qué sucede si aplicamos Head Tail l? Recordemos que la asociatividad a la izquierda haría algo así Head(Tail) l pero Tail en ese momento no es nada, y si aplicamos head explota. En este caso los paréntesis son importantes: (head (tail l))

Veamos otro ejemplo $map(\backslash x \rightarrow x\ 0)\ (map(+)\ [1, 2, 3])$. En este caso la asociatividad a la izquierda nos muestra algo así $map(\backslash x \rightarrow x\ 0)\ ((map(+)\ [1, 2, 3]))$. Esto genera $map(\backslash x \rightarrow x\ 0)\ [2, 3, 4]$ ahora según lo que haga la función x , hace una cosa u otra. Imaginemos que lo llamamos como $map\ (+1)\ [2, 3, 4]$ esto daría $[3, 4, 5]$

Función \$

Aplica una función a un valor

1		$g :: (a \rightarrow b) \rightarrow (a \rightarrow b)$
2		$g\ x\ y = x\ y$

Construyendo una lista paso a paso con constructores

Ej.: ¿Como construimos la lista $[1, 2]$ utilizando constructores?

- Lo primero que necesitamos, es un constructor de listas. Para eso tenemos la expresión $(:)$. Recordando que la aplicación de expresiones es asociativo a izquierda.
- Veamos su tipo desde GHCi $(:) :: a \rightarrow [a] \rightarrow [a]$.
- Necesitamos enviarle un valor de tipo a , una lista y como resultado, la operación $(:)$ devuelve una lista.
- Preguntemos lo siguiente: ¿Con $((:) 1)$ nos basta para agregar a una lista? No, porque no estamos cumpliendo el tipado del constructor. Si quisiéramos una lista con solamente el 1 si bastaría, pero acá también queremos el 2.
- Entonces, comenzamos aplicando la expresión $(:)$ con el número 2, y ahí sí enviamos como segundo parámetro una lista vacía (que da como resultado) una lista vacía.
- $(((:) 2)\ []) = [2]$
- Por último, $((:) 1)\ [2]$ también cumple el tipo pues nos quedaría $((:) 1)\ [2] = [1, 2]$

¿Y si quisiéramos construir 1, 2, 3, 4? Recordemos la asociatividad a la izquierda, pero a la hora de querer utilizar una expresión, hay que cumplir el tipado. Hasta que ese tipado no se cumpla, Haskell tratará de resolver la expresión más profunda para ver si reduciendo cumple el tipo.

$((:1)\ (((:) 4)\ (((:) 2)\ (((:) 3)\ []))))$

Esto lo podemos ver como $a \rightarrow (b \rightarrow (c \rightarrow (d)))$ pues para conocer a , necesitamos construir la lista de la derecha, para conocer a b necesitamos la lista de la derecha, para conocer a c necesitamos la lista de d , y d inicializa la lista vacía. Esto es importantísimo porque claramente no podríamos usar $(:) 1$ sin una lista. Entonces Haskell evalúa todo lo de la derecha hasta que obtenga una lista. Si no se obtuviera una lista, sería inválido.

Polimorfismo

Sucede en aquellas expresiones que tienen más de un tipo.

- $id :: a \rightarrow a$
- $(:) :: a \rightarrow [a] \rightarrow [a]$

- $fst :: (a, b) \rightarrow a$
- $cargarStringArray :: String \rightarrow \square \rightarrow [String]$. No es una expresión Polimórfica.

Nota: Es importante que si tenemos 2 parámetros de tipo a significa que los dos parámetros deben ser de ese tipo. Si tenemos 2 parámetros uno de tipo a y uno de b podría suceder que sean diferentes pero puede que sean el mismo. **Importante:** Recordar que si usamos operadores, colocar las clases que correspondan. Ej.: $a > 0$ a debe ser ($Num a, Orda$). Véase anexo para ver ejemplos de las clases de tipos.

Polimorfismo con Clases (Type Classes)

Es posible limitar el polimorfismo a clases específicas. Es decir, si nos mandan un tipo a podemos decir que ese tipo a es genérico pero de una clase específica.

Ejemplo: $func :: Num a \Rightarrow a \rightarrow a \rightarrow a$

En este caso, $Num a$ es una Type Class.

Modelo de Cómputo (cálculo de valores)

Dada una expresión, se computa su valor usando las ecuaciones siempre y cuando estén bien tipadas.

Importante: Que una expresión se cumpla el tipado, no significa que devuelva un valor.

- $sumarUno :: a \rightarrow a$: No falla nunca.
- $division :: a \rightarrow b$: Falla si $b = 0$. Esto nos demuestra que aunque los parámetros estén bien tipados, podemos tener indefiniciones.

¿Cómo está dado un programa en Funcional?

Un programa funcional está dado por un conjunto de **ecuaciones orientadas**. Se les llama de esta forma pues del **lado izquierdo está lo que define y del lado derecho la definición** (o expresión que produce el valor) Una ecuación $e1 = e2$ se interpreta desde dos puntos de vista

- **Denotacional:** Declara que $e1$ y $e2$ tienen el mismo significado.
 - "Denotan lo mismo"
- **Operacional:** Computar el valor de $e1$ se reduce a computar el valor de $e2$.
 - "Operan de la misma forma"

¿Cómo es el lado izquierdo de una ecuación orientada?

No es una expresión arbitraria. Debe ser una función aplicada a **patrones**.

Un patrón puede ser:

- Una variable: a, b, c .
- Un comodín: $_$.
- Un constructor aplicado a patrones: Recordemos que un constructor sería algo que construye un tipo.
 - $True, False, 1$, etc.

Importante: El lado izquierdo NO debe contener variables repetidas.

Ej.: $iguales\ x\ x = True$ es una expresión mal formada. Esto pues dos valores que pueden ser diferentes, no pueden caer en la misma variable.

Ej.: $predecesor\ (n + 1) = n$ también está mal formada porque estamos haciendo un cálculo del lado izquierdo, y recordemos que del lado izquierdo solo hay definiciones. Del lado derecho se hacen los cálculos o cálculo de los valores.

¿Cómo evaluamos las expresiones?

- Buscamos la subexpresión más externa que coincida con el lado izquierdo de una ecuación.
- Reemplazar la subexpresión que coincide con el lado izquierdo de la ecuación por la expresión correspondiente al lado derecho.
- Continuar evaluando la expresión resultante.

¿Cuándo se detiene la evaluación de una expresión?

- Cuando el programa estalla.
 - Loop infinito.
 - Indefinición.
- Cuando la expresión es una función **parcialmente aplicada**. ¿qué sería esto? ¿se refiere a utilizar mal la función parcialmente aplicada?
- La expresión es un constructor o un constructor aplicado. Ej.: True, (:) 1, [1, 2, 3].
 - Yo lo veo como algo más del tipo: una fórmula atómica o algo irreducible.

¿Cómo ayuda el Lazy Evaluation (Evaluación Perezosa) a Haskell?

Muchas veces nos ayuda a evitar tocar con un valor indefinido aunque esté ahí. Como no evalúa cosas que no necesita, si hay un indefinido por ahí y no necesita ni siquiera llegar, no lo toca.

```
1 |   indefinido :: Int
2 |   indefinido = indefinido
3 |   head(tail [indefinido, 1, indefinido])
```

¿Qué hace tail? Toma la cola de la lista, entonces como resultado arroja [1, indefinido] (ni siquiera evaluó el primer indefinido)
¿Qué hace head ahora entonces? [1] (ni siquiera evaluó el indefinido del final)

Importancia del orden de las Ecuaciones

El criterio más fuerte debe ir por encima del resto. Porque puede haber un caso que matchee y nunca se evalúe el siguiente caso.

Veamos un ejemplo:

```
1 |   esCorta (_:_:_) = False
2 |   esCorta _ = True
```

Considerando este orden:

- Si mando una lista que siempre tiene ≥ 3 elementos da False.
- Si mando una lista que tiene < 3 elementos es siempre True.

Cambiamos el orden y veamos qué cambia

```
1 |   esCorta _ = True
2 |   esCorta (_:_:_) = False
```

Considerando este orden:

- Si la lista tiene 0, 1, 2, 3, 4, ... n elementos siempre dará True.

En la segunda aplicación ¡nunca caemos en el caso 2! y esto es importante porque sabemos que si tiene 1 cae siempre en la primera.

Notación Infija & Notación Prefija

Infija: argumento + funcion + argumento

Prefija: funcion + argumentos

Importante: $(\leq)18 \neq (\leq 18)$ pues la opción de $(\leq)18$ sería $(18 \leq x)$ mientras que (≤ 18) sería $(x \leq 18)$. Donde claramente denota que el parámetro cuando está fuera de la notación infija es el primer argumento mientras que si lo ponemos adentro hardcodeamos su posición. Es decir $(18 \leq)$ estaríamos diciendo $(18 \leq x)$

Curricación

Una función es curificada cuando tiene la siguiente forma $ab :: Int \rightarrow Int \rightarrow Int$

Esto puede parecer que recibe 2 parámetros y devuelve uno, pero en realidad lo que hace es básicamente por cada parámetro asociar a la izquierda y hacer una función por cada parámetro.

Es decir, sería algo así $ab :: Int \rightarrow (Int \rightarrow Int)$

La función Curry **NO CAMBIA** la manera en que la función original curificada recibe los argumentos, sino que, la función curry es una especie de *punto* para que nosotros mandemos los argumentos separados y los aplique a la función no

currificada.

Importante: No tiene sentido hablar de currficación cuando las funciones tienen un solo parámetro. Las funciones currificadas son realmente importantes en la Programación Funcional porque nos permite aplicarlas de forma parcial. Es una especie mas reutilizable.

Una función no currificada tiene esta pinta $\text{suma} :: (\text{Int}, \text{Int}) \rightarrow \text{Int} \rightarrow \text{Int}$ porque acá estoy obligando a la función suma a recibir una tupla de elementos, no puedo ir mandando de a uno parcialmente. Véase **anexo** para armar una función curry y uncurry

Funciones de Orden Superior

Son funciones que reciben como parámetro otras funciones.

Definamos la composición de funciones ($g \circ f$)

Recordemos que en Álgebra vimos Composición de Funciones y es algo así: sean $A : B \rightarrow C$ y $D : A \rightarrow B$

La composición gof es $A \rightarrow B$. Es decir, va desde el dominio de D hasta la imagen de A. (la salida de una función debe ser la entrada de la otra.)

En Haskell, la podemos definir así

```
1  |   ent: Entrada sal: Salida
2  |
3  |   (.) :: (b -> c) -> (a -> b) -> (a -> c)
4  |
5  |   Queremos: (g . f) x ==> g (f x)
6  |
7  |   Desglosemos
8  |   (.) :: (ent. de g -> sal. de g) -> (ent. de f -> sal. de f) -> (ent. de f -> sal. de g)
9  |
10 |   El resultado de la composición nos da una nueva función que tiene como entrada un valor de tipo a, y la
    |   salida es un valor de tipo c.
11 |
12 |   Entonces, primero se evalúa f x, lo cual produce un resultado de tipo b.
13 |
14 |   Luego, este resultado (de tipo b) se pasa a g, produciendo un resultado de tipo c.
15 |
16 |   (g . f) x: envía el parámetro x a la función f, y el resultado de f x se manda a g. Esto da como resultado
    |   g (f x)
```

Otra forma de definir la composición es usando Notación Lambda.

```
1  |   (.) :: (b -> c) -> (a -> b) -> (a -> c)
2  |   g . f = \ x -> g (f x)
```

Recordemos que esto es algo recordable pues fog sería meter g adentro de f y para eso, la salida de g debe ser la entrada de f. Y luego se devuelve como salida el dominio de g y la salida de f. Nótese que la notación lambda es súper útil para definir funciones sin nombre, que solamente hagan algo. Esto es muy útil cuando queremos mandar una función por parámetro y que una función dada la llame a esta función.

Llamados estándar vs llamados en composición

$\text{not}(\text{null}x) \equiv (\text{not.null})x$

Esto quiere decir que si primero componemos todas las funciones que queremos aplicarle a un valor, es lo mismo que ir aplicando de una a una las funciones al argumento. **Importante:** La composición funciona **sí y solo sí** alguna de las funciones está parcialmente aplicada. Ej.: $(== 0).\text{mod}nm$ no funciona, pero $(== 0).\text{mod}n$ sí.

Funciones Lambda

Son funciones anónimas, es decir, no tienen nombre.

$(\backslash x \rightarrow x + 1)$: Es una función anónima que dado un x, te devuelve una función que le aplica x+1.

Funciones Lambda Anidadas

$\backslash y \rightarrow (\backslash x \rightarrow y)$: Esto a ojo es una función constante, porque dado un valor y, se aplica la función x pero se devuelve el mismo valor y.

Los parámetros se obvían en algunos casos, este es uno. La mejor forma sería $\lambda y \rightarrow \lambda x \rightarrow y$.
Mejor notación para funciones anidadas: $\lambda y x \rightarrow y$

Reducción de Lambda

Preguntar $foldr(\lambda x rec \rightarrow fx : rec) \equiv \lambda x \rightarrow (:)(fx)$ y $\lambda x \rightarrow ex \equiv e$

Asignar nombre a una función

$a = \lambda x \rightarrow x$: al nombre **a** le asigno la función anónima $\lambda x \rightarrow x$

¿Para qué queremos funciones de orden superior? Pt 1

```
1 | dobleL :: [Float] -> [Float]
2 | dobleL [] = []
3 | dobleL (x:xs) = x * 2 : dobleL xs
4 |
5 | esParL :: [Int] -> [Bool]
6 | esParL [] = []
7 | esParL (x:xs) = x `mod` 2 == 0 : esParL xs
```

¿Qué es lo que tienen en común? Todas tienen una estructura bastante similar.

```
1 | g [] = []
2 | g (x : xs) = f x : g xs
```

Lo único que cambia es **qué se hace** en cada paso recursivo.

Hagamos una pregunta ¿la cantidad de elementos de la entrada es igual a la de la salida? en este caso sí pero ¿qué operación se está haciendo? se están haciendo ciertas manipulaciones de los datos y se devuelve la información modificada en una lista nueva.

Esto, en varios lenguajes se conoce como **map**. Se hace una manipulación de los datos pero se devuelve **la misma cantidad de elementos**.

Map

Recibe como parámetro una función que es aplicada a todos los elementos y devuelve una lista nueva. Se utiliza para modificar los valores de una lista dada según lo que haga la función.

```
1 | map :: (a -> b) -> [a] -> [b]
2 | map f [] = []
3 | map f (x : xs) = f x : map f xs
```

Entonces, podemos definir **qué operación de modificación** se le realizan a los elementos de una lista dada.

```
1 | multiplicarPorDos :: [a] -> [b]
2 | multiplicarPorDos xs = map (\x -> x * 2) xs
3 |
4 | dividirPorDos :: [a] -> [b]
5 | dividirPorDos xs :: map (\x -> x / 2) xs
```

Entonces gracias a Map nos abstraemos de tener miles de funciones con la misma estructura pero solo cambien **qué hacen** con los elementos.

Véase **anexo** para ver ejercicios interesantes con Map.

¿Para qué queremos funciones de orden superior? Pt 2

```
1 | negativos :: [Int] -> [Int]
2 | negativos [] = []
3 | negativos (x:xs) = if x < 0
4 |                     then x : negativos xs
5 |                     else negativos xs
6 |
7 | pares :: [Int] -> [Int]
8 | pares [] = []
9 | pares (x:xs) = if x `mod` 2 == 0
```

```

10 |         then x : pares xs
11 |         else pares xs

```

¿Qué es lo que tienen en común? Todas tienen una estructura bastante similar, pero lo único que cambia es **cuando vamos a agregar a la lista los elementos**.

```

1 | g [] = []
2 | g (x:xs) = f x : g xs

```

Hagamos una pregunta ¿la cantidad de elementos de la entrada es igual a la de la salida? Puede que sí, puede que no. ¿Qué operación se está haciendo? se están filtrando ciertos elementos de una lista que no cumplan una condición dada. Esto, en varios lenguajes se conoce como **filter**. Se devuelve una nueva lista con los elementos que cumplan una condición dada.

Filter

Recibe como parámetro una función que es aplicada a todos los elementos. Se utiliza para "borrar" elementos de una lista, o quitar aquellos que no cumplan un criterio dado.

```

1 | filter :: (a -> Bool) -> [a] -> [a]
2 | filter p [] = []
3 | filter p (x : xs) = if p x
4 |                       then x : filter p xs
5 |                       else filter p xs

```

La función que le enviamos es para especificar **qué elementos nos queremos quedar**.

Entonces, podemos definir **qué operación de filtro** se le realizan a los elementos de una lista dada.

```

1 | eliminarImpares :: [a] -> [a]
2 | eliminarImpares xs = filter (\x -> x `mod` 2 == 0) xs
3 |
4 | borrarNegativos :: [a] -> [a]
5 | borrarNegativos xs = filter (\x -> x > 0) xs

```

Entonces gracias a Filter nos abstraemos de tener miles de funciones con la misma estructura pero solo cambien **con qué elementos nos quedamos** dada una condición

Recursión

¿Qué tienen en común los siguientes problemas? ¿En qué difieren?

```

1 | concat :: [[a]] -> [a]
2 | concat [] = []
3 | concat (x:xs) = x ++ concat xs
4 |
5 | reverso :: [a] -> [a]
6 | reverso [] = []
7 | reverso (x:xs) = reverso xs ++ [x]
8 |
9 | sum :: [int] -> int
10 | sum [] = 0
11 | sum (x:xs) = x + sum xs
12 |
13 | En común:
14 |     1. Todos tienen un caso base, pero es diferente.
15 |     2. Todos hacen un paso recursivo, pero cambia la operación que hacen.
16 |     3. El tipo de entrada puede no coincidir con el de la salida.

```

En este tipo de problemas lo mejor que podemos hacer es realizar una especie de función que nos permita modularizar lo más posible y aquello que difiere, pasarlo por parámetros.

Este tipo de problemas lo podemos solucionar con Foldr.

Recursión Estructural (foldr)

Se la conoce como Plegado a la Derecha porque va anidando el llamado de la función hacia adentro hasta que llega al caso base. Cuando llega al caso base, resuelve de derecha a izquierda. Una función f está dada por recursión estructural sí:



- El caso base devuelve un valor fijo z.
- El caso recursivo es una función de x y (g xs)
- Se trabaja solamente con la cabeza de la lista.
- Se hace recursión sobre la cola pero no se tiene acceso a ella, sino al llamado recursivo. Es decir, el caso recursivo no usa xs.
- La recursión es la clásica, va desde derecha a izquierda. La R de foldr es de Right.

Su estructura formal es la siguiente

```

1 | g [] = <caso base> -> valor
2 | g (x:xs) = <caso recursivo> -> función cabeza de la lista y resultado de la recursión.

```

Por lo tanto foldr se define como

```

1 | foldr :: (a -> (b -> b)) -> b -> [a] -> b
2 | foldr f z [] = z
3 | foldr f z (x:xs) = f x (foldr f z xs)

```

Importantísimo: El tipado de foldr es $Foldable\ t \implies (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow ta \rightarrow b$

- a = Es el elemento que tenemos actualmente en la recursión.
- b = Es cualquier tipo que nos permita acumular. Puede ser una tupla, un número, lo que sea. Pero en cada paso recursivo hay que llenar eso. El caso base tiene que cumplir este tipo b.
- Véase [anexo](#) para ver ejemplos usando foldr.

¿Qué es lo que produce foldr (: [])? Es una identidad sobre listas porque haría recursión sobre una lista vacía.

¿Qué es lo que hace la siguiente función foldr? $foldr(\backslash x\ r \rightarrow x)\ 0\ [1, 2]$

- Toma la lista y hace recursión. Empieza con el 2, ejecuta la función y devuelve 2. Hace el paso recursivo.
- Toma la lista y hace recursión. Ahora sigue con el 1, recibe 1 y la salida es 1.
- Esta función agarra el primer elemento de una lista, sería el head.

Importante: Foldr puede trabajar con listas infinitas.

Véase anexo para ver ejemplos de Foldr.

Recursión que no es estructural

```

1 | ssort :: Ord a => [a] -> [a]
2 | ssort [] = []
3 | ssort (x:xs) = minimo (x:xs) : ssort(sacarMinimo(x:xs))

```

No es estructural pues se esta usando xs para el llamado de minimo(x:xs) y no solamente en el llamado recursivo de ssort.

Iteración (foldl)

Se la conoce como Plegado a la Izquierda porque va resolviendo inmediatamente de izquierda a derecha hasta que termine el proceso.

Este tipo de recursión es más que recursión una iteración, en este caso empezamos yendo desde el primer valor hasta el último.

En este enfoque voy modificando una solución parcial.

Por lo tanto foldl se define como

$$\text{foldl1 } f \text{ ac } \left(\begin{array}{c} (:) \\ / \quad \backslash \\ 1 \quad (:) \\ / \quad \backslash \\ 2 \quad (:) \\ / \quad \backslash \\ 3 \quad [] \end{array} \right) \rightsquigarrow^* \left(\begin{array}{c} f \\ / \quad \backslash \\ f \quad 3 \\ / \quad \backslash \\ f \quad 2 \\ / \quad \backslash \\ \text{ac} \quad 1 \end{array} \right)$$

```

bin2dec :: [Int] -> Int
bin2dec = foldl1 (\ ac b -> b + 2 * ac) 0

bin2dec [1, 0, 0]
  ~> foldl1 (\ ac b -> b + 2 * ac) 0 [1, 0, 0]
  ~> foldl1 (\ ac b -> b + 2 * ac) (1 + 0) [0, 0]
  ~> foldl1 (\ ac b -> b + 2 * ac) (0 + 2 * (1 + 0)) [0]
  ~> foldl1 (\ ac b -> b + 2 * ac) (0 + 2 * (0 + 2 * (1 + 0))) []
  ~> 0 + 2 * (0 + 2 * (1 + 0))
  ~> 4

```

```

1 | foldl :: (b -> a -> b) -> b -> [a] -> b
2 | foldl f ac [] = ac
3 | foldl f ac (x:xs) = foldl f (f ac x) xs

```

¿Qué es lo que sucede en el siguiente ejemplo?

```

1 | foldl (\x y -> 1) 0 unos
2 | foldl f (f 0 1) unos
3 | foldl f(f(f 0 1) 1) unos

```

Esto es in-realizable con foldl. Es decir, foldl no puede manejar listas infinitas.

Importantísimo: El tipado de foldl es $Foldable\ t \implies (b \rightarrow a \rightarrow b) \rightarrow b \rightarrow ta \rightarrow b$

- b = Es cualquier tipo que nos permita acumular. Puede ser una tupla, un número, lo que sea. Pero en cada paso recursivo hay que llenar eso. El caso base tiene que cumplir este tipo b .
- a = Es el elemento que tenemos actualmente en la recursión.

Véase [anexo](#) para ver ejemplos usando foldl.

¿Por qué foldl es peor que foldr?

foldl: Procesa la lista de izquierda a derecha, lo que requiere evaluar toda la lista antes de devolver un resultado, lo que no es posible con listas infinitas.

foldr: Procesa la lista de derecha a izquierda, permite trabajar de manera perezosa y puede manejar listas infinitas si la función y el valor inicial permiten una evaluación parcial o completa sin necesidad de procesar todos los elementos.

Recursión Primitiva (recr)

No existe en Haskell. Es una manera de nosotros podemos tener las mismas ventajas de foldr pero en este caso, la recursión primitiva nos permite utilizar la cola de la lista

- Se trabaja solamente con la cabeza de la lista y la cola.
- Se hace recursión sobre la cola.

Su estructura forma les la siguiente

```

1 | g [] = b
2 | g (x:xs) = f x xs (g xs)

```

Por lo tanto recr se define como

```

1 | recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
2 | recr f z [] = z
3 | recr f z (x:xs) = f x xs (recr f z xs)

```

Tipos

Existen diferentes maneras de definir tipos. Esto es según sea el objetivo. Definimos tipos con la palabra **data** + Nombre = Tipo1 — Tipo2 — Tipo 3

Si hacemos en GHCI : *tTipo1* saldrá que es de tipo Nombre.

Nota: | indica que a continuación hay otro tipo.

Tipos Comunes

Son no recursivos, ej: $data Dia = Lu|Ma|Mi|Mie|Ju|Vi|Sa|Do$

Lu, Ma, Mi, ... son constructores del tipo dia.

Los argumentos podrán ser recibidos diciendo qué tipo se espera, y usamos pattern matching para utilizarlos.

```
1 |   EsLunes :: Dia -> Bool
2 |   EsLunes Lu = True
3 |   EsLunes _ = False
4 |
5 |   EsFinDeSemana :: Dia -> Bool
6 |   EsFinDeSemana Sa = True
7 |   EsFinDeSemana Do = True
8 |   EsFinDeSemana _ = False
```

Tipos con Funciones

Los tipos también pueden tener tipos que necesiten argumentos. Difieren en la info que devuelven.

Ej.: $data Persona = LaPersona String String Int$

Importante

- Si evaluamos $:tLaPersona$ sin enviar los argumentos retornará $LaPersona :: String -> String -> Int$.
- Si evaluamos $:tLaPersona "T", "H", 23$ enviando los argumentos retornará que LaPersona es de tipo Persona

```
1 |   Edad :: Persona -> Int
2 |   Edad (LaPersona n a e) = e
3 |
4 |   Cumpleaños :: Persona -> Persona
5 |   Cumpleaños (LaPersona n a e) = LaPersona n a (e+1)
```

Importante: Nótese que estamos devolviendo **una nueva persona**. En programación funcional no existe el concepto de "modificar.algo, sino crear algo nuevo con lo anterior y cambiarle algo.

Tipos Recursivos

Cuando tengo tipos recursivos, las funciones que los usen deben manejar casos bases y la recursión

Ej.: $data Nat = Zero \text{---} Succ Nat$

Tipos Polimórficos

Al igual que las funciones, podemos definir que los tipos tengan constructores de un tipo específico.

```
1 |   data List a = Vacía | Const a (List a)
```

Importante en Tipos

No se puede repetir un mismo constructor para un mismo tipo.

Es decir

```
1 |   data List = Vacía | Cons Int ListI
2 |   data List a = Vacía | Cost a (List a)
3 |
4 |   Error. No puede estar Vacía como constructor de dos tipos diferentes.
```

Listas

- Por extensión: Es dar la lista implícita escribiendo todos sus elementos. Ej. $[1, 2, 3]$
- Secuencias: Progresiones aritméticas en un rango particular. Ej.: $[3..7]$ es la lista que tiene los números del 3 al 7.
- Por compresión: Se definen de la siguiente manera $[expresion \text{---} selectores, condiciones]$. Ej.: $[(x,y) \text{---} x \text{ i- } [0..5], y \text{ i- } [0..3], x+y==4]$ es la lista de pares que tienen elementos de x e y que dan menos que 4

Estructuras Recursivas sobre Otras Estructuras de Datos

¿Cómo podemos plantear la estructura de un foldr para un árbol binario o cualquier estructura plegable que no sea una lista?

Lo primero que podemos intuir es que en las listas, solo hay una recursión, sobre la lista propiamente pero en un Árbol Binario tenemos dos caminos: rama izquierda y rama derecha. Esto quiere decir que de alguna manera, tenemos que capturar 2 recursiones.

- Planteamos varios problemas acerca de esa estructura de datos y vemos qué patrones hay en común.
- Las cosas que sean diferentes, las pasamos por parámetros.

Por ejemplo, sean estas operaciones de un árbol binario

```
1 |   nodos :: AB -> Int
2 |   nodos Nil = 0
3 |   nodos (Bin i r d) = nodos i + 1 + nodos d
4 |
5 |   preorder :: AB -> [a]
6 |   preorder Nil = []
7 |   preorder (Bin i r d) = [r] ++ preorder i ++ preorder d
```

Podemos observar que cambia lo siguiente

- Tipos de salida: En el primer ejemplo devolvemos un int, en el segundo una lista de a. La lista de a sería el tipo que tenga el AB.
- Caso base: En uno es [] y en otro 0. Por lo tanto tenemos que admitir un tipo b para el caso base. El caso base es del mismo tipo que el tipo de salida **siempre**.
- Función que realiza: En uno hace sumas, en el otro hace concatenaciones.

¿Cómo podríamos escribir las funciones anteriormente mencionadas de manera anónima?

```
1 |   nodos: (\ri r rd -> ri + 1 + rd)(nodos i) r (nodos d)
2 |   preorder: (\ri r rd -> [r] ++ ri ++ rd) (preorder i) r (preorder d)
```

Recordemos la firma de foldr: $Foldable\ t \implies (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow t\ a \rightarrow b$

Donde b era el caso base / recursivo y a el elemento actual, acá tenemos dos.

Recordemos la estructura del tipo AB: $AB\ a = Nil \mid Bin\ (AB\ a)\ a\ (AB\ a)$

Entonces nuestra función foldAB va a tener que estar preparado para recibir ambos (AB a) y a. $foldAB :: (b \rightarrow a \rightarrow b \rightarrow b) \rightarrow b \rightarrow AB\ a \rightarrow b$ donde $t\ Bin: (AB\ a) \rightarrow a \rightarrow (AB)\ a \rightarrow AB\ a$ donde

- $(b \rightarrow a \rightarrow b \rightarrow b)$
 - la primera b: recursión rama izquierda
 - la a: la raíz
 - la segunda b: recursión rama derecha
 - la tercera b: el resultado del proceso.
- b: el resultado.
- AB a: la entrada
- b: el resultado.

Es súper importante entender que la firma de la función que acepta foldAB está prácticamente atada al tipo de la estructura de dato.

Validación y Verificación de Programas

- Trabajaremos con estructuras de datos **finitas**. Más técnicamente, con tipos de datos **inductivos**.
- Trabajamos con **funciones totales**
 - Las ecuaciones deben cubrir todos los casos posibles.
 - La recursión siempre termina.
- El programa **no depende del orden** de las ecuaciones.

Principio de Reemplazo

Sea $e1 = e2$ una ecuación del programa. Las siguientes operaciones preservan la igualdad de expresiones.

- Reemplazar **cualquier instancia** de $e1$ por $e2$.
- Reemplazar **cualquier instancia** de $e2$ por $e1$.

Importante: Si una igualdad se puede demostrar usando sólo el principio de reemplazo, decimos que la igualdad vale **por definición**.

```
{L0} length [] = 0
{L1} length (_ : xs) = 1 + length xs
{S0} suma [] = 0
{S1} suma (x : xs) = x + suma xs

Veamos que length ["a", "b"] = suma [1, 1]:

      length ["a", "b"]
    = 1 + length ["b"]      por L1
    = 1 + (1 + length [])  por L1
    = 1 + (1 + 0)          por L0
    = 1 + (1 + suma [])    por S0
    = 1 + suma [1]         por S1
    = suma [1, 1]          por S1
```

Inducción Estructural

Inducción sobre booleanos

Anexo

Recordando Haskell

Para ejecutar un archivo hay que instalar GHCi. Una vez instalado, nos paramos en la terminal en el directorio donde está el archivo que queremos ejecutar.

- Cargar archivo: `:l nombreArchivo`
- Ver tipo: `:type tipo`
- Ejecutar funcion: `funcion parametro1 parametro2...`
- Recargar archivo: `:r`
- Si necesitamos hacer cálculos para mandar un parámetro, usar paréntesis: Ej.: `otherwise = n * factorial(n-1)`

Maybe

El Maybe se utiliza en Haskell para recibir/devolver respuestas condicionales que pueden ser de un tipo u otro.

Se define como *data Maybe a = Nothing | Just a*

Ej.: *devolverFalsoSiVerdadero : Bool → Prelude.Maybe Bool*

El Maybe deja la puerta abierta a un valor posible "Nothing". Entonces tenemos dos casos: Si me envían un True devuelvo False (tipo bool), caso contrario, devuelvo Nothing.

Either

El Either se utiliza en Haskell para poder recibir/devolver un parámetro que podría ser de un tipo u otro.

Se define como *data Either a b = Left a | Right b*

Para poder saber qué operación hacer según el tipo literalmente en código usamos (Left valor) o (Right valor).

Ej.: *devolverRepresentacionIntBool :: Either Int Bool → Int*

Si es un entero, devuelvo ese mismo entero porque no hago nada. Eso lo hacemos con $Left(a) = a$, ahora, si el tipo es booleano tengo que decir explícitamente la respuesta según su valor. Es decir, $Right(False) = 0$ sino, $Right(True) = 1$.

Declaración de tipos en Haskell

Se utiliza `data nombretipo tipo = Tipo 1 | Tipo 2 El` | se interpreta como **o bien**

Árboles Binarios

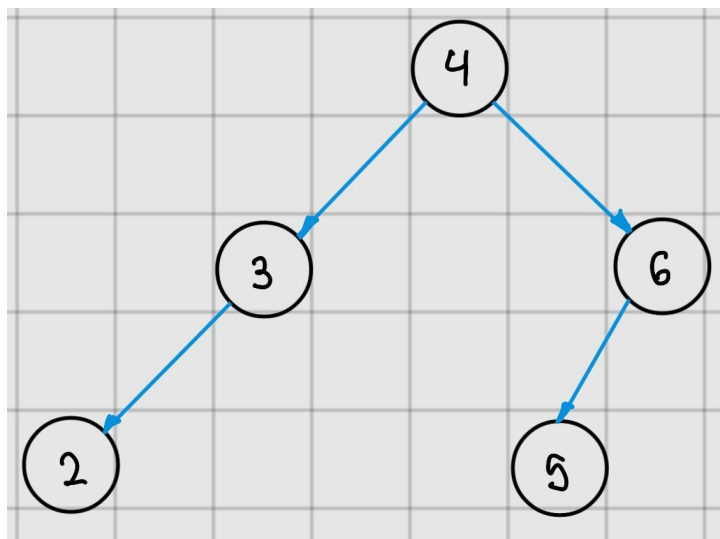
Es un tipo (para mi parecer) meramente recursivo.

`data AB a = Nil | Bin (AB a) a (AB a)` Nótese que es algo re contra recursivo, porque para definir el tipo de AB a decimos que es un Bin que a su vez es de AB a y a su vez AB a es otro árbol binario. Veamos unos ejemplos de esto

- `Bin (Nil) Nil (Nil)`: es el árbol que no tiene ni siquiera raíz. Y nótese que en cada paréntesis es importante indicar el Nil pues es la forma de que el tipado de Haskell nos lo acepte.
- `Bin (Bin Nil 3 Nil) 4 (Bin Nil 6 Nil)`: Es el árbol que comienza con un Nodo raíz que tiene el valor de 4. El hijo izquierdo del Nodo con valor 4 es otro árbol binario que tiene como valor 3 en su nodo y no tiene hijos. El hijo derecho del Nodo con valor 4 es otro árbol binario que tiene como valor 6 en su Nodo y no tiene hijos.

Y así sucesivamente, veamos un dibujo para tener algo más visual.

El siguiente árbol binario: `Bin (Bin (Bin Nil 2 Nil) 3 Nil) 4 (Bin (Bin Nil 5 Nil) 6 Nil)` representa el siguiente:



Curry & Uncurry

Digamos que necesitamos currificar una función que recibe una tupla de elementos. Es decir, algo así: $suma :: (Int, Int) \rightarrow Int$

Por la definición de curry necesitamos que por cada argumento, haya una función que lo devuelva, por lo tanto el resultado sería algo así $suma :: Int \rightarrow Int \rightarrow Int$.

Veamos el tipo de función que queremos currificar: $((a, b) \rightarrow c)$, esto lo queremos llevar a $a \rightarrow b \rightarrow c$.

Por lo tanto nuestra función curry sería algo así:

```

1 |   curryOwn :: ((a, b) -> c) -> a -> b -> c
2 |   curryOwn f a b = f (a, b)

```

Entonces, digamos que queremos hacer la suma currificada.

```

1 |   sumTuple :: (Float, Float) -> Float
2 |   sumTuple (x, y) = x + y
3 |
4 |   sumarCurry :: Float -> Float -> Float
5 |   sumarCurry = curryOwn sumTuple

```

Lo que hace `sumarCurry` es llamar a `curry(sumTuple)` es decir, a `curry` le manda la función `sumTuple`. Los parámetros que le mandamos a `sumarCurry` como a `-i b`, los convierte en `(a, b)` para poder aceptar el tipo de la función `sumTuple`.

¿Cómo sería entonces la función `uncurry`? Si recibimos los argumentos en forma de $a \rightarrow b \rightarrow c$ debo llevarlo a $(a, b) \rightarrow c$

```

1 |   uncurryOwn :: a -> b -> c -> ((a, b) -> c)
2 |   uncurryOwn f (a, b) = f a b
3 |
4 |   sumarUncurry :: (a, b) -> c
5 |   sumarUncurry = uncurryOwn sumarCurry

```

Esto quiere decir que vamos a llamar a `sumarUncurry` que recibe la tupla, ahora `sumarCurry` está currificada, por lo que la tenemos que convertir nuevamente a la función no currificada, para luego llamar a `sumTuple` de la manera original.

Clases de Tipos

- `Num a`: Indica que el parámetro `a` es numérico
- `Ord a`: Indica que el parámetro `a` es ordenable bajo algún criterio, es decir, podemos aplicar `>` `<` `=` etc.
- `Eq a`: Indica que el parámetro `a` se puede igualar, es decir, podemos aplicar `=`

Foldr

```

1 |   // Solo recorre listas de tipo a. Es decir, devuelve la suma de los elementos.
2 |   sumFoldrlist :: Num a => [a] -> a
3 |   sumFoldrlist = foldr (\x ac -> x + ac) 0
4 |
5 |   // Recorre tipos plegables. Aquí no nos limitamos solo a listas, porque véase que usamos t a en vez de [a]
6 |   sumFoldr :: (Foldable t, Num a) => t a -> a
7 |   sumFoldr = foldr (\x ac -> x + ac) 0

```

Flip

Toma dos parámetros y devuelve una función que los devuelve en el orden inverso.

Es decir: $(a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$

Luego: $flip\ f\ a\ b = f\ b\ a$

Función identidad

Devuelve el mismo valor aplicado a la función.

Es decir: $(a \rightarrow b) \rightarrow a \rightarrow b$

Luego: $\$f\ a = f\ a$

Función constante

Devuelve un valor enviado sin aplicarle ninguna función.

Es decir: $a \rightarrow b \rightarrow a$

Luego: $const\ a\ b = a$

Reduciendo expresiones elegantemente

- $filter(\backslash x \rightarrow length\ x > 3)$
 - ¿Puede hacerse algo mejor? No. Porque a `x` si o sí necesitamos aplicarle una función.
- $filter(\backslash x \rightarrow x > n)$
 - ¿Puede hacerse algo mejor? Sí. $filter(> n)$
- $filter(\backslash x \rightarrow mod\ x\ 2 / = 0)$
 - ¿Puede hacerse algo mejor? No. Porque a `x` le tenemos que calcular su módulo con 2.
- $map(\backslash x \rightarrow map(\backslash y \rightarrow toUpper\ y)\ x)$
 - ¿Puede hacerse algo mejor? Primero entendamos que hace, recorre una lista de palabras, luego en cada palabra toma cada letra y la pasa a mayúscula. Esto es un doble `map`, uno por palabra otro por letra. Entonces sí $map\ (map\ toUpper)$
- $doblar\ Elementos.\ filtrar\ Pares$
 - ¿Puede hacerse algo mejor? No. Esto es el equivalente a un lenguaje imperativo `doblarElementos(filtrarPares(lista))`

¿Qué hacen las siguientes funciones compuestas?

```
1 flip($) 0 id
2
3 (==0) . (flip mod 2)
4
5 Primero veamos que hace flip mod 2.
6 mod 2 es notación infija (Integral a => 2 -> a -> a), entonces lo que está diciendo es que si le paso
7 cualquier número va a hacer mod 2 x, y nosotros por lo que yo entiendo es que queremos ver si es par.
8 Por lo tanto, lo primero que haríamos es invertir los argumentos de mod 2 con flip (Integral a => a -> 2
9 -> a), entonces quedaría algo como mod x 2 donde el x lo tenemos que enviar nosotros.
10 Luego, se compone la función de mod x 2 == 0 esperando solo un argumento donde verifica si efectivamente
11 un número dado es par.
12 Entonces, (Integral a => x -> Bool)
13
14 map f = ((:) . f)
15 Lo que hace esta función es básicamente aplicar una función f a todos los elementos y agregarlos a una
16 lista particular.
17
18 Dado ["hola", "abc"] quiero devolver ["cba", "aloh"]. Es decir, dar vuelta cada caracter de cada palabra y
19 ademas dar vuelta las palabras.
20
21 reverseAnidado :: ["String"] -> ["String"]
22 reverseAnidado = reverse . (map . reverse)
23 Lo primero que hacemos es hacer un map haciendo reverse por cada caracter de la lista. Luego, reordenamos
24 las palabras en sí.
25 El tipo de reverse es: [a] -> [a] pero con los elementos al revés. Entonces, por cada palabra (map)
26 hacemos un reverse y las guardamos.
27 Finalmente, nos queda algo así ["aloh", "cba"], nos queda dar vuelta eso, entonces hacemos nuevamente un
28 reverse de toda la lista. ["cba", "aloh"].
29
30 listacomp f xs p = [f x | x <- xs, p x]
31 listacomp f xs p = map f (filter p xs)
```

Ejercicios Foldr

1. Realizar lo mismo que el ejercicio hecho con (Foldl) pero el último elemento menos el anteúltimo, más el ante ante último, etc.

En este caso es obvio que conviene usar foldr porque empezamos desde el último mandando como caso base a (0, True) para colocar mínimo el primer elemento.

Vamos a tener que hacer también el tema de fst para terminar sacando la suma al revés.

Por lo tanto

- El caso base es (0, True)
- La recursión la vamos a hacer con $\backslash x(acc, sign)$
- Ejemplo con [1, 2, 3]
 - $x = 3, acc = 0, sign = + \rightarrow acc = 3$
 - $x = 2, acc = 3, sign = - \rightarrow acc = 1$
 - $x = 1, acc = 1, sign = + \rightarrow acc = 2$

Ejercicios Foldl

1. Defina la función sumaAlt, que realiza la suma alternada de los elementos de una lista. Es decir, primer elemento menos el segundo mas el tercero menos el cuarto y así.

Lo primero que notamos es que de alguna manera vamos a tener que ver paso a paso si tenemos que sumar o restar. Esto lo podemos hacer de una manera simple, deberíamos en cada paso recursivo mandar la orden de si sumar o restar para poder usarlo en un condicional.

Entonces nuestro paso recursivo además de llevar un acumulador va a llevar un booleano. Por lo tanto el caso base inicial podemos usar una especie de tupla como (0, True).

Por otro lado, notamos que si usamos foldr haría el procedimiento al revés, por lo tanto tenemos que usar foldl para garantizar el desarrollo de izquierda a derecha.

Veamos el tipado de Foldl


```

1 | foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
2 | Lo primero que recibe es un tipo b que en nuestro caso será (Int, Bool).
3 | Luego a es el elemento que va cambiando en cada paso recursivo.

```

Importante notar que si queremos devolver un número, lo que va a devolver siempre el paso recursivo es algo igual al caso base (0, True) pero a nosotros no nos piden eso sino que nos piden que enviemos el número.

Por lo tanto, luego de aplicar foldl podemos aplicar fst para obtener el acumulador.

```

1 | sumaAlt :: (Num a) => [a] -> a
2 | sumaAlt = fst . foldl (\(acc, isAdd) x -> if(isAdd) then (acc+x, False) else (acc-x, True)) (0, True)

```

2. Definir la función sumasParciales que dada una lista de números devuelve otra de la misma longitud que tiene en cada posición la suma parcial de los elementos de la lista original desde la cabeza hasta la posición actual.

Entendamos el enunciado:

- Vamos a usar foldl para ir sumando de izquierda a derecha.
- El tipado de foldl es $b \rightarrow a \rightarrow b$ donde b es nuestro primer argumento acumulador y a el elemento.
- Necesito de alguna manera tener el valor inmediato anterior. Podemos hacer algo como empezar enviando una lista vacia como caso base, y a medida que vamos haciendo la recursion tomar la cabeza de la lista.
- Si la lista esta vacia entonces solo agrego x a la lista de la recursion (primer elemento), si no esta vacia sumo con el elemento de la lista (cabeza)
- Porque foldl labura así $[1, 2, 3]$
 - $1:[] = [1]$
 - $2 + (\text{head } [1]) : [1] = [3, 1]$
 - $3 + (\text{head } [3, 1]) : [3, 1] = [6, 3, 1]$
 - Ahora podemos usar reverse y el resultado es $[1, 3, 6]$

Entonces la solución sería algo así:

```

1 | sumasParciales :: Num a => [a] -> [a]
2 | sumasParciales = reverse . foldl (\acc x -> if(length acc > 0) then x+(head acc):acc else x:acc) []

```

Ejercicios Map

1. Realice una función mapDoble que toma una función currificada de dos argumentos y dos listas de igual longitud y devuelve una lista de aplicaciones de la función a cada elemento correspondiente de las dos listas.

Básicamente $f = x + y$ $l1 = [1, 2]$ $l2 = [3, 4]$ da como resultado $[4, 6]$

Desglosemos el ejercicio en partes

- 1. Lo primero que necesito hacer básicamente es recorrer ambas listas de alguna manera a la vez, y obtener algo como $[(1, 3), (2, 4)]$ y luego aplicar a esa lista de pares la función f. Si nos ponemos a pensar, basta con hacer una función que reciba dos listas de tipo $[a]$ y $[b]$ y devuelva una lista de $[(a, b)]$.
- 2. Una vez que tenemos esta lista de pares, sabemos que la función que nos va a enviar tiene que utilizar ambos elementos a la vez, pero la función es de la forma $a \rightarrow b \rightarrow c$ y esto quiere decir que está currificada pero nuestra lista de pares es de tipo $[(a, b)]$ por lo tanto antes de aplicar f debemos aplicar *uncurry* f lista para que cuando mandemos f y la lista, f se convierta en una función que espere (a, b) .
- 3. Por último, para aplicar a todos los elementos de la lista podemos usar map de la siguiente forma: *map (uncurry f) (lista)*

```

1 | mapDobleCorta :: (a -> b -> c) -> [a] -> [b] -> [c]
2 | mapDobleCorta f l r = map (uncurry f) (armarPares l r)

```

donde la función armarPares tiene la siguiente pinta

```

1 | armarPares :: [a] -> [b] -> [(a, b)]
2 | armarPares _ [] = []
3 | armarPares [] _ = []
4 | armarPares (x:xs) (y:ys) = (x, y) : armarPares xs ys

```

2. Realice una suma de matrices.

- Idea: Necesitamos de alguna manera recorrer la fila 1 de la matriz 1 y la fila 1 de la matriz 2. Esto lo podemos hacer facilmente reutilizando el ejercicio anterior (armarPares), es decir, enviamos armarPares con la fila1 matriz1 y fila1 matriz2. Esto nos armaría los pares de esa fila.

- Tenemos que generalizar este proceso para cada fila. Por lo tanto podemos recibir una matriz, y podemos hacer recursion sobre cada lista de la matriz.
- A su vez, vamos a necesitar que una vez que tenemos los pares armados, sobre esos pares se aplique un map haciendo uncurry sobre f. Porque si tenemos $F1\ M1 + F1\ M2 = [(1, 2), (3, 4), (5, 6)]$ esto indica que la fila de la M1 es $[1, 3, 5]$ y la fila de la M2 es $[2, 4, 6]$. Por lo tanto, lo que necesito hacer es convertirlo en $[[9, 12]]$ y agregarlo a la lista resultante. El Uncurry acá es ultra importante porque mi función pide $Int \rightarrow Int \rightarrow Int$ y yo voy a mandar $(Int, Int) \rightarrow Int$.

```

1 sumaMat :: (Int -> Int -> Int) -> [[Int]] -> [[Int]] -> [[Int]]
2 sumaMat _ [] _ = []
3 sumaMat _ _ [] = []
4 sumaMat f (x:xs) (y:ys) = map (uncurry f) (armarPares x y) : sumaMat f xs ys

```

¿Qué es lo que podríamos cambiar? Estamos haciendo un laburo exactamente igual mapDoble.

```

1 sumaMat :: (Int -> Int -> Int) -> [[Int]] -> [[Int]] -> [[Int]]
2 sumaMat f = mapDoble(mapDoble(f))

```

Armando funciones que permitan hacer recursión sobre un tipo dado

1. **foldNat**: Necesitamos hacer recursión sobre los números enteros. Una excelente pregunta es ¿recursión sobre números naturales?. Sí.

Un número natural se define de la siguiente manera $data\ Nat = Zero \mid Succ\ Nat$. Es decir, tiene dos chances: O es cero, o es un sucesor de algún número.

¿Cuántos casos tendríamos que probar si quisieramos verificar la correctitud del tipo? 2. Que sea Zero o que sea algún sucesor. Así, de esta manera, podemos definir al número 4 como $Succ(Succ(Succ(Succ\ Zero)))$.

La recursión nos sirve justamente para esto, para poder hacer operaciones con números naturales.

Ej.: Necesitamos multiplicar un número n m veces ¿Cómo hacemos esto? Sumamos el mismo número m veces. Es decir, si quiero hacer $n * n$ equivale a decir $n+n+n+n+n$. Entonces ¿Cómo podríamos hacer esto?

Para empezar, pensemos en qué tipo de operaciones queremos hacer con foldNat. Podríamos hacer multiplicación, potencia, etc.

Pensemos por un momento ¿qué pasaría si el caso base fuese 0 si estamos sumando? Nada, porque justamente para la multiplicación la queremos hacer como $n+n+n+n+n$ y si llegamos a Zero quiero que devuelva 0.

Ahora ¿qué sucede si queremos hacer la potencia? Recordemos que la potencia se define como la multiplicación de un número m veces. Si nos abstraemos a nuestro esquema $n^m \equiv n * n * n * n...m \equiv (n + n) + (n + n) + (n + n) + (n + n)...m\ veces$ si quisieramos aplicar el caso base de 0 para la multiplicación se nos haría 0. Por lo tanto tenemos otro caso base, sería 1.

Por lo tanto, definamos el foldNat pero utilizando el tipo de Integer (como pidió la cátedra)

```

1 foldNat :: Integer -> (Integer -> Integer) -> Integer -> Integer
2 foldNat base _ Zero = base
3 foldNat base f n = f (foldNat base f (n-1))

```

Entonces ahora podemos definir la multiplicación como

```

1 multiplicacion :: Integer -> Integer -> Integer
2 multiplicacion n m = foldNat 0 (+n) m

```

¡Nótese que acá el caso base es 0 porque estamos sumando!

Por último podemos definir la potencia reutilizando la multiplicación

```

1 potencia :: Integer -> Integer -> Integer
2 potencia n m = foldNat 1 (multiplicacion n m) m

```

Es importante notar que la potencia requiere hacer foldNat nuevamente porque tenemos que hacer el proceso de multiplicación m veces.

2. **Polinomios**: Hablemos ahora de Polinomios