Práctica 1: Programación Funcional

Tomás Felipe Melli

July 7, 2025

Índice

1 Currificación y Tipos

En esta práctica no está permitido el uso dela **recursión explícita**(como la implementación de factorial que se llama a sí misma (factorial 0 = 1 , factorial n = n * factorial (n-1))).
La **Currificación** en Haskell es un proceso de transformación que ocurre cuando una función que toma múltiples argumentos se convierte en una secuencia de funciones en donde cada una toma sólo un argumento. Esta es la forma en la que el lenguaje interpreta a las funciones con múltiples argumentos : como una serie de funciones que aceptan uno solo. Miremos la siguiente función :

```
suma :: Int -> Int -> Int
suma a b = a + b
```

Que escrita :

```
suma :: Int -> (Int -> Int)
suma a b = a + b
```

Es equivalente y sucede siempre, ya que **en Haskell todas las funciones están currificadas.** Esto induce a mencionar el concepto de **Partial Application** que es el proceso de aplicar algunos (pero no todos) los argumentos de una función currificada, creando una nueva con los argumentos restantes. Miremos el ejemplo con la función suma

```
sumaUno = suma 1
```

Donde **sumaUno** es el resultado de *aplicar parcialmente* la función *suma*. En realidad es una nueva función que toma un *Int*, le suma 1 y devuelve el resultado. La propiedad que se cumple en este ejemplo es que el operador − > es **asociativo a derecha** y la **aplicación de la función es a izquierda** donde la signatura de la función luce como ya vimos :

```
suma :: Int -> (Int -> Int)
```

Que significa que la función **suma** toma un argumento, y devuelve una función que toma otro y devuelve un Int.

1.1 Ejercicio 1

1.1.1 a) Hay que dar el tipo de las siguientes funciones

```
max2 (x, y)
  | x >= y    = x
  | otherwise = y

normaVectorial (x, y) = sqrt (x^2 + y^2)

subtract = flip (-)

predecesor = subtract 1

evaluarEnCero = \f -> f 0

dosVeces = \f -> f . f

flipAll = map flip

flipRaro = flip flip
```

En el caso de *max2* podemos ver que recibe una tupla y compara cuál de los dos valores es mayor. En Haskell se usa el **operador => de firma de tipo** para separar las **restricciones de los tipos de los parámetros**. **Ord** es una **clase de tipos**, contratos que los tipos deben cumplir, en el caso de la clase Ord : **define tipos que pueden ser comparados entre sí y los tipos que pertenecen a esta clase deben proporcionar una implementación para estas funciones de comparación**. Con esto en mente, los elementos que forman parte de la tupla, que sean del tipo que quieran, pero deben pertenecer a la clase de tipos Ord. Ese tipo tendrá definida la operación >= y la función podrá devolver un resultado correcto.

```
max2 :: (Ord a) => (a,a) -> a

(Float,Float) -> Float
```

El caso de *normaVectorial*, si miramos el **type** de sqrt desde ghci :

```
ghci> ::type sqrt
sqrt :: Floating a => a -> a
```

Sigamos mirando los tipos que necesita cada operador ...

```
ghci> ::type (^)
(^) :: (Num a, Integral b) => a -> b -> a
ghci> ::type (+)
(+) :: Num a => a -> a -> a
```

Con esto en mente, a priori, **x** e **y** tienen que ser de la clase Num por la operación de potenciación. Coincidente con la clase de tipo necesaria en la posterior suma. Finalmente, *sqrt* exige que el valor sea del tipo que quiera mientras pertenezca a la clase de tipo **Floating**. Dicho esto, como la clase de tipo Num engloba : **Integral (tipos : Int, Integer), Fractional(subclase : Floating) Real**, y por tanto, Floating, es subclase de una subclase de Num, la restricción recae sobre esta. Concluimos que la signatura para *normaVectorial* es:

```
normaVectorial :: (Floating a) => (a,a) -> a

(Float,Float) -> Float
```

Veamos la función *substract* que hace uso de **flip** y **(-)**

```
ghci> ::type flip
flip :: (a -> b -> c) -> b -> a -> c
ghci> ::type (-)
(-) :: Num a => a -> a -> a
```

Que básicamente, invierte el orden de los dos argumentos de la función (-).

```
substr :: (Num a) => a -> a -> a
substr x y = x - y
substract = flip substr
----------------------
ghci> substr 4 2
2
ghci> substract 4 2
-2
```

Entonces el tipo de esta función será ...

```
substract :: (Num a) => a -> a -> a
```

Pero como nos pide suponer que los numeros son de tipo Float, tenemos que modificar la clase de tipo a **Fractional**

```
Float -> Float -> Float
```

Para el caso de la función *predecesor* que hace uso de *substract*, la signatura es

```
predecesor :: (Fractional a) => a -> a

Float -> Float
```

La función **evaluarEnCero** lo que hace es utilizar la expresión para **aplicar funciones de orden superior** al valor 0. En este caso, la función es **f**, es una **función anónima que toma una función como parámetro** y luego aplica al valor, en este caso, 0.

```
func :: (Num a) => a -> a
func x = x + 10

evaluarEnCero = \f -> f 0
----------------------
ghci> evaluarEnCero func
10
```

Dicho esto, el tipo será

```
func :: (Num a) => a -> a
func x = x + 10

evaluarEnCero :: (Num a) => (a -> a) -> a

(Float -> a) -> a
```

Esto es así porque la función **f** recibe como parámetro una función, que sabemos que es **func** que recibe un parámetro de tipo a y devuelve uno de tipo a. Como resultado, devuelve el valor. Por ello, la signatura es correcta.
En el siguiente punto, vemos la **composición de funciones** $(f\ o\ g)(x) = f(g(x))$

```
dosVeces = \f -> f . f
```

Por la forma en que está definida **f**, lo que sucede es que $(f\ o\ f)(x) = f(f(x))$ ya que la composición es con sí misma. Por tanto, **dosVeces** recibe cierta función **f** y la compone con sí. Recordamos **func** y hacemos la prueba :

```
func :: (Num a) => a -> a
func x = x + 10
----------------------
ghci> dosVeces func 10
30
```

Entonces, el tipo de **dosVeces** responde al hecho de que recibe una función y luego un parámetro para ella y retorna un resultado.

```
ghci> ::type dosVeces
dosVeces :: (a -> a) -> a -> a
```

flipAll está definida como sigue :

```
flipAll = map flip
----------------------
ghci> ::type map
map :: (a -> b) -> [a] -> [b]
```

Ya hablamos un poco de **map** en la P0, pero nunca está mal reforzar. Si vemos la signatura, map lo que hace es tomar una función que toma un valor de tipo a y retorna uno de tipo b y se aplica sobre los elementos de una lista que son de tipo a y finalmente retorna una lista donde los elementos son de tipo b.

```
ghci> map (+1) [1,2,3]
[2,3,4]
```

Ahora bien, si a map le pasamos flip, va a necesitar, como dijimos, elementos de una lista de tipo a. Como flip toma funciones, los elementos deben ser efectivamente funciones. Pasemos en limpio la situación : map recibe una función y una lista de elementos, flip una función que toma dos parámetro para invertirles el orden. **flipAll** entonces necesita recibir una lista de funciones.

```
funciones = [(/),(-)]
flipAll = map flip
----------------------
ghci> map (\f -> f 4 2) (flipAll funciones)
[0.5,-2.0]
```

A modo de ejemplo, queremos que aplique la función f con los parámetros 4 y 2 que es un elemento del array de funciones que ya fue flipeada. Entonces, ahora, la división está definida como b *'div'* a y por ello es *0.5* y análogo a la resta.
Finalmente, el tipo de la función es, una lista de funciones y retorna la lista de funciones flipeadas.

```
ghci> ::type flipAll
flipAll :: [a -> b -> c] -> [b -> a -> c]
```

La función **flipRaro** lo que hace es invertirle los parámetros a flip, esto es, si flip naturalmente es

```
ghci> ::type flip
flip :: (a -> b -> c) -> b -> a -> c
```

Lo que sucederá es que reordena la forma en que flip recibe los parámetros de manera que ahora quede así:

```
b -> (a -> b -> c) -> a -> c
```

Es decir, le pasamos un parámetro, la función y el otro parámetro para que **flipRaro** funcione correctamente, como sigue :

```
ghci> flipRaro 4 (-) 2
-2
```

1.1.2 b) Indicar cuáles no están currificadas y definirle su forma currificada. Dar nuevamente el tipo

Para la primer función, como ya mencionamos su tipo, es trivial notar que no se puede construir una serie de funciones que toman un sólo argumento cada una ya que esta toma una tupla de una y retorna, haciendo imposible la aplicación parcial. Si quisiésemos currificarla :

```
max2 :: (Ord a) => a -> a -> a
max2 x y| x >= y = x
        | otherwise = y
```

Para la siguiente función, **normaVectorial** también podemos sacarle que tome la tupla y los tome los parámetros por separado para currificarla.

```
normaVectorial :: (Floating a) => a -> a -> a
normaVectorial x  y  = sqrt ( x ^2 + y ^2)
```

Un ejemplo interesante, porque la función **predecesor** daría la sensación de no estar currificada, pero en realidad es justo parte de la serie de funciones en la aplicación parcial que hace que substract lo sea.
Todas las demás están currificadas.

1.2 Ejercicio 2

Tenemos que definir dos funciones, una para **currificar** una función no-currificada y otra que descurrifica una currificada.

```
curry :: ((a,b) -> c) -> (a -> b -> c)
curry f x y = f (x,y)

uncurry :: (a -> b -> c) -> ((a,b) -> c)
uncurry f (x,y) = f x y
```

1. Para el caso de **curry**, lo que sucede es que, toma una función de tipo *unaria* (toman una dupla como parámetro) y luego dos parámetros x e y. Curry entonces convierte a dupla esos parámetros y luego aplica la función unaria.
2. En el caso de **uncurry**, lo que sucede es, se toma una función *binaria* y una dupla. Uncurry entonces lo que hace es desarticular la dupla en dos parámetros separados y aplica la función a ellos.
3. No se podría definir esa función ya que para dar el tipo habría que conocer N. Necesitaríamos transformar :

   ```
   curryN :: ((a,b,c,d,...) -> z) -> a -> b -> c -> d -> ... -> z
   ```

   Pero en Haskell no hay un tipo tupla que tome una cantidad arbitraria de argumentos.

2 Esquemas de recursión

Si miramos las implementaciones de **map** y **filter** :

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


filter' :: (a -> Bool) -> [a] -> [a]
filter' p [] = []
filter' p (x:xs)| p x == True = x : filter' p xs
                | otherwise = filter' p xs
```

Vamos a notar cierta similitud en la forma en que se organiza el esquema recursivo. Es por ello que podemos generalizar este esquema con la función **foldr** (fold rigth) porque toma una lista y la **reduce** desde la derecha. Como sucede en las funciones que acabamos de ver. Supongamos map' :

$$map' \ (+1) \ [1,2,3]$$
$$(1 + (2 + (3 + 1)))$$

Es decir, que se **foldea** la lista de derecha a izquierda. Dicho esto, veamos **foldr**:

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

Con esta implementación en mente, lo que hace la función **foldr** es tomar una función, un parámetro y una lista. Qué pasa con la signatura ? La función foldr, como foldea de derecha a izquierda, necesita que la función que toma como parámetro (que por lo que vemos es **binaria**) tome uno de sus parámetros por consola y debe, obligatoriamente devolver algo de tipo **b**. Esto es porque el tipo de ese dato debe ser coincidente con el valor que quedará en la lista semifoldeada. Veamoslo desglosado :

$$foldr' \ (+) \ 1 \ [1,2,3]$$

Se aplica la función suma con el parámetro 1. Algo como (+1). Toma algo de tipo a y devuelve algo de tipo b en este caso :

$$(1 + (2 + (3 + 1)))$$

como 3 es un elemento de la lista y es de tipo a, debe retornar efectivamente algo de tipo b, ya que el resultado de ir foldeando la lista debe concidir con la signatura de esta '(+)'. El elemento de tipo a, lo saca de la lista, el de tipo b, es el parámetro que le pasamos. Y así, hasta foldearla completamente...

$$(1 + (2 + (4)))$$
$$(1 + (6))$$

y devolver el elemento de tipo b, como se ve a continuación:

$$= 7$$

Antes de pasar a resolver el ejercicio, vamos a investrigar un poco sobre esto de **Foldable.**
En Haskell, **Foldable** es una clase de tipo en el que se definen un conjunto de estructuras de datos que **se pueden reducir a un único valor mediante una operación de plegado (fold)**.

```haskell
class Foldable t where
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldl :: (b -> a -> b) -> b -> t a -> b
    foldMap :: Monoid m => (a -> m) -> t a -> m
    -- otros m todos de la clase...
```

En este caso **t** representa un tipo de **contenedor**, como una lista, un árbol,... que pueda contener elementos de tipo a. Como ya vimos **foldr**, también existe **foldl** que hace la reducción desde izquierda a derecha. Otro ejemplo de funciones con esta clase de tipos es **foldMap** que toma una función que mapea un valor de tipo a a un valor de tipo m (donde m es un Monoid), y luego pliega los resultados de esa función sobre los elementos del contenedor. Esto es útil para combinar valores de una estructura de datos que tienen un monoid de tipo. Esto es ... ?

```haskell
class Monoid m where
    mempty  :: m  -- El elemento identidad del monoid
    mappend :: m -> m -> m  -- Operaci n binaria que combina dos valores de tipo m
```

Un **tipo algebraico** que representa un conjunto con una operación binaria que satisface tres propiedades fundamentales: asociatividad, elemento identidad, y cerradura. No nos vamos a meter en mucho detalle con esto por ahora pero, veamos rápidamente :

1. **Cerradura**: La operación binaria (mappend) toma dos elementos del conjunto y produce un tercer elemento que también pertenece al mismo conjunto.
2. **Asociatividad**: La operación binaria debe ser asociativa. Es decir, para cualquier a, b, y c del conjunto, debe cumplirse que:

   ```haskell
   (a `mappend` b) `mappend` c == a `mappend` (b `mappend` c)
   ```

3. **Elemento identidad:** Debe existir un elemento neutro (mempty) tal que para cualquier elemento a del conjunto:

   ```haskell
   mempty `mappend` a == a
   a `mappend` mempty == a
   ```

En fin, volvamos ...

2.1 Ejercicio 3

2.1.1 I

Nos piden, usando foldr, redefinir : sum, elem, (++), filter y map :

- **sum :** lo que hace es calcular la suma de los elementos de una estructura de clase Foldable

  ```haskell
  ghci> sum [1,2,3,4]
  10
  ```

  Para implementar **sum** con foldr, primero conviene pensar en :

  ```haskell
  sum :: (Foldable t, Num a) => t a -> a
  sum [ ] = 0
  sum (x:xs) x + sum xs
  ```

  Por tanto, con foldr

  ```haskell
  sum' :: (Num a) =>[a] -> a
  sum' lista = foldr (+) 0 lista
  ---------------------
  ghci> sum' [1,2,3,4]
  10
  ```

- **elem :** verifica si un elemento está en una estructura. Forma parte de la clase Foldable. SU signatura es

  ```haskell
  elem :: (Foldable t, Eq a) => a -> t a -> Bool
  ```

  Que si pensamos en ...

  ```haskell
  elem x [ ] = false
  elem x (y:ys) = x == y || elem x ys
  ```

  Que con foldr...

```
elem' :: (Eq a) => a -> [a] -> Bool
elem' n = foldr (\x rec -> (x == n) || rec) False
---------------------
ghci> elem' 3 [4..7]
False
ghci> elem' 3 [3..6]
True
```

Veamos cómo foldea la lista de manera de obtener este resultado
$[4,5,6,7] \rightarrow \lambda(4,\ \lambda(5,\ \lambda(6,\ \lambda(7, False))))$
$\lambda(4,\ \lambda(5,\ \lambda(6\ , \lambda(7, False))))$
$\lambda(4,\ \lambda(5,\ \lambda(6\ , False)))$
$\lambda(4,\ \lambda(5,\ False))$
$\lambda(4, False)$
$= False$
Segundo caso $[3,4,5,6] \rightarrow \lambda(3,\ \lambda(4,\ \lambda(5,\ \lambda(6, False))))$
$\lambda(3,\ \lambda(4,\ \lambda(5\ , \lambda(6, False))))$
$\lambda(3,\ \lambda(4,\ \lambda(5\ , False)))$
$\lambda(3,\ \lambda(4,\ False))$
$\lambda(3, False)$
$= True$

- **(++) :** esta función concatena listas en una sola. Lo que queremos lograr es algo de este estilo :

$$(++)\ [1,2]\ [3,4] \rightarrow (1 : (2 : [3,4])) = [1,2,3,4]$$

```
concatenar' :: [a] -> [a] -> [a]
concatenar' xs ys = foldr (\x rec -> x : rec) ys xs
---------------------
ghci> concatenar' [1,2] [3,4]
[1,2,3,4]
```

En simples términos, lo que hace la λ es tomar un parámetro x y **rec** que es la lista que viene de la recursión. Este x lo agrega a rec. **ys** y **xs** están en ese orden ya que, miremos la signatura de foldr nuevamente :

```
foldr :: (a -> b -> b) -> b -> [a] -> b
```

La función es la λ el parámetro al cuál queremos concatenarle cierta lista es **ys** pero el primer parámetro es el que nos pasan, **xs**

- **filter :** la función filter ya vimos lo que hace, ahora la implementamos usando foldr.

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

---------------------
ghci> map'' (+1) [1,2,3]
[2,3,4]
```

- **map :**

```
map'' :: (a -> b) -> [a] -> [b]
map'' f = foldr (\x rec -> f x : rec) []
---------------------
ghci> filter'' (>1) [1..10]
[2,3,4,5,6,7,8,9,10]
```

En este caso se introduce algo interesante, el uso de la estructura **if .. then .. else**. La λ recibe un **x** y la lista que se está reduciendo en **rec**. El predicado que nos interesa evaluar por parámetro **p** y la lista sobre la cual queremos hacer la comparación también. En cada paso vemos el x y decidimos si la agregamos a la reduccuión o no. Los parámetros de foldr son la λ, la lista vacía [ ] (sobre la cual vamos a ir agregando digamos los valores que cumplen), y la lista que tiene todos los elementos que queremos filtrar **xs**.

2.1.2 II

Hay que implementar, usando foldr, la función **mejorSegún** que básicamente lo que hace es generalizar la aplicación de una función a una estructura foldeable para saber cuál es el mejor elemento de manera que sólo tengamos que definir qué es ese criterio.

```haskell
mejorSegun :: (a -> a -> Bool) -> [a] -> a
mejorSegun p [x] = x
mejorSegun p (x:xs) = if p y (mejorSegun p xs) then y else mejorSegun p xs
----------------------
-- con foldr --
mejorSegun' p = foldr (\y rec -> if p y rec then y else rec)
----------------------
ghci> mejorSegun (>) [1,23,4,100]
100
ghci> mejorSegun (<) [1,23,4,100]
1
```

2.1.3 III

Nos piden que dada una lista de números, devolvamos otra de la misma longitud y en cada posición, dar la suma acumulada desde la cabeza hasta la posición actual. Luego de varios intentos de que funcione con foldr, existe una que se llama **foldl**. Su signatura es :

```haskell
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
```

Y su utilidad, como foldr es reducir una estructura Foldable, en este caso, aplicando **una función binaria de manera acumulativa de izquierda a derecha**. El primer parámetro de tipo **b** es el **acumulador (acc)**, el segundo es el del tipo de la estructura **a**. Es decir, **b** es el valor inicial del acumulador. Es del tipo b y es lo que foldl usará para empezar a hacer la reducción de la lista.

$$
\begin{aligned}
&foldl\ (+)\ 0\ [1,2,3,4] \\
&-> (((0+1)+2)+3)+4 \\
&-> ((1+2)+3)+4 \\
&-> (3+3)+4 \\
&-> 6+4 \\
&-> 10
\end{aligned}
$$

Como se ve acá, el **acc** va aumentando desde el primer elemento hasta el último en la lista.

```haskell
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' fn acc [] = acc
foldl' fn acc (x:xs) = foldl' fn (fn acc x) xs
```

Con esto en mente, la idea es ...

```haskell
sumasParciales :: (Num a) => [a] -> [a]

sumasParciales = foldl (\acc x -> if null acc then x:acc else acc ++ [last acc + x]) []
```

Vamos recorriendo la lista... **x** es nuestro *actual* en ese recorrido. El **acumulador es y**(ojo : en este escenario no es un número, es una lista). Si **y** $==$ [], estamos al principio, ponemos lo que tenga **x** (**[x]**), caso contrario, queremos el valor de la suma acumulada hasta el momento, y eso lo sacamos con **last**. A ese valor, le sumamos el x, el actual y tenemos que concatenar este elemento a la lista acumulada.

2.1.4 IV

En este enunciado hay que foldear una lista de manera que :

$$
\begin{aligned}
&sumaAlt[1,2,3] \\
&= 1-(2-(3-0)) \\
&= 2
\end{aligned}
$$

Primero lo pensamos :

```haskell
sumaAlt :: Num a => [a] -> a
sumaALt [] = 0
sumaALt (x:xs) = x - sumaAlt xs
```

Si lo pensamos como funciona el foldr, que va foldeando la lista de derecha a izquierda, el (-) al resolver los operadores, nos da algo como :

```haskell
sumaAlt :: [Int] -> Int
sumaAlt = foldr' (-) 0
----------------------
ghci> sumaAlt [1,2,3]
2
```

2.1.5 V

Lo mismo pero

$$\begin{aligned} & sumaAlt'[1,2,3] \\ &= 3-(2-(1-0)) \\ &= 0 \end{aligned}$$

Primero lo pensamos :

```
sumaAlt2 :: Num a => [a] -> a
sumaAlt2 = sumaAlt2' 0

sumaAlt2' ac [] = ac
sumaAlt2' ac (x:xs) = sumaAlt2' (x-ac) xs
```

Con foldl

```
sumaAlt2 = foldl (flip (-)) 0
```

2.2 Ejercicio 4

2.2.1 I

Nos recomiendan usar **concatMap**. Lo bueno de esta función es que **aplica una función a cada elemento de una lista y luego concatenar todas las listas resultantes en una sola lista.** La signatura es ...

```
concatMap :: (a -> [b]) -> [a] -> [b]
```

Un ejemplo de uso sería, donde dentro del corchete le ponemos una serie de funciones para aplicar, primero que sume uno, después que sume 2, ... finalmente, toma los resultados y concatena las listas:

```
ghci> concatMap (\x -> [x + 1, x + 2]) [1,2,3]
[2,3,3,4,4,5]
```

Solución :

```
insertar :: a -> [a] -> [[a]]
insertar x xs = [take i xs ++ [x] ++ drop i xs | i <- [0..length xs]]
```

```
permutaciones :: [a] -> [[a]]
permutaciones [] = [[]]
permutaciones (x:xs) = concatMap (insertar x) (permutaciones xs)
```

Con foldr

```
permutaciones = foldr (\x rec -> concatMap (insertar x) rec) [[]]
```

2.2.2 II

```
partes :: [a] -> [[a]]
partes [] = [[]]
partes (x:xs) = map (x:) (partes xs) ++ partes xs
```

con foldr

```
partes = foldr (\x rec -> map (x:) rec ++ rec) [[]]
```

2.2.3 III

```
prefijos :: [a] -> [[a]]
prefijos [] = [[]]
prefijos (x:xs) = map (x:) (prefijos xs) ++ [[]]
```

con foldr

```
partes = foldr (\x rec -> map (x:) rec ++ [[]]) [[]]
```

2.2.4 IV

```
sublistas : [a] -> [[a]]
sublistas [] = [[]]
sublistas (x:xs) = sublistas xs ++ (map ((:) x) (prefijos xs))
```

con foldr no se puede hacer

```
sublistas xs = foldr (\x rec -> sublistas xs ++ (map ((:) x) (prefijos xs)) [[]]
```

porque la f no necesita sólo rec, o sea sublistas xs, sino también xs para calcular prefijos xs. O sea, no es recursión estructural, es **primitiva**. Por tanto, usamos **recr**

```
sublistas  = recr (\x xs rec -> (map (x:) (prefijos xs)) ++ rec )[[]]
```

2.3 Ejercicio 5

Piden decir si la recursión es estructural o no. En caso de serlo, reescribirla con *foldr*. Caso contrario, explicar el motivo

```
elementosEnPosicionesPares :: [a] -> [a]
elementosEnPosicionesPares [] = []
elementosEnPosicionesPares (x:xs) =
    if null xs
        then [x]
        else x : elementosEnPosicionesPares (tail xs)
entrelazar :: [a] -> [a] -> [a]
entrelazar [] = id
entrelazar (x:xs) =
\ys -> if null ys
        then x : entrelazar xs []
        else x : head ys : entrelazar xs (tail ys)
```

1. Para el caso de **elementosEnPosicionesPares** no es recursión estructural ya que **xs no es parámetro de su propio llamado recursivo**. Además, el llamado recursivo se hace sobre un argumento que no es xs sino tail xs.

2. La función **entrelazar** es recursión extructural porque sólo usa xs como parte de un llamado recursivo, y estos los hace sólo pasando xs como argumento. Por dicho motivo, podemos escribirla con foldr:

   ```
   entrelazar :: [a] -> [a] -> [a]
   entrelazar = foldr (\x rec -> \ys -> if null ys then x : rec [] else x : head ys: (rec (tail ys))
       ) id
   ```

2.4 Ejercicio 6

En este ejercicio se introduce la **recursión primitiva**.

2.4.1 sacarUna

Esta función recibe un elemento y una lista y devuelve el resultado de eliminar la primer aparición. Queremos pasarle a recr una función que recibe **x xs rec**. Con esa info, miramos si **x == elemento que nos pasan**. En dicho caso, devolvemos la cola de (x:xs). Pero sino, queremos que x se agregue al llamado recursivo.

```
recr :: (a -> [a] -> b -> b) -> b -> [a] -> b
recr _ z [] = z
recr f z (x:xs) = f x xs (recr f z xs)

sacarUna :: Eq a => a -> [a] -> [a]
sacarUna _ [] = []
sacarUna elem (x:xs) = if x == elem then xs else x:(sacarUna elem xs)
----------------
con recr :
sacarUna a = recr (\x xs rec -> if x == a then xs else x : rec) []
```

2.4.2 Por qué no sirve foldr en este caso ?

No es correcto foldr ya que sólo permite que la función f use como parámetro al llamado recursivo sobre xs. Sin embargo, en este caso, también necesitamos que tome a xs como parámetro, ya que podríamos usarlo en el caso en que $x == elem$ para cortar la recursión.

2.4.3 insertarOrdenado

Queremos insertar al elemento que nos pasan de forma que quede ordenado crescientemente en la lista ordenada que nos pasan.

```
insertarOrdenado :: Ord a => a -> [a] -> [a]
insertarOrdenado elem [] = [elem]
insertarOrdenado elem (x:xs) = if x < elem then x:(insertar elem xs) else elem : x : xs
```

Escrito con recr

```
insertarOrdenado elem = recr (\x xs rec -> if x < elem then x : rec else elem : x : xs) [elem]
```

2.5 Ejercicio 7

2.5.1 mapPares

Esta función es una versión de map que aplica una función currificada a una lista de pares de valores y devuelve una lista de aplicaciones de la función a cada par. En otras palabras, recibe una función **f currificada**, una lista de tuplas **[(a,b)]** y queremos devolver una lista con las tuplas evaluadas por esa función. Dónde está el chiste ? Tenemos algo como :

```
sumar :: Int -> Int -> Int
sumar x y = x + y
mapPares sumar [(1, 2), (3, 4), (5, 6)]
```

Y la idea es que se sumen entre sí y devuelvan el resultado de evaluarse.

```
mapPares :: (a -> b -> c) -> [(a,b)] -> [c]
mapPares f = map (uncurry f)
---------------------
ghci> mapPares (+) [(1,2), (3,4), (5,6)]
[3,7,11]
```

2.5.2 armarPares

Esta función es **zip**

```
armarPares [] = const []
armarPares (x:xs) = \ys -> if null ys then [] else (x,(head ys)) : armarPares xs (tail ys)
```

con foldr

```
armarPares xs = foldr (\x rec ys -> if null ys then [] else (x, head ys) : rec (tail ys)) (const [])
```

Acá rec es una función que arma pares entre la cola de xs y lo que se le pasa como parámetro.

2.5.3 mapDoble

```
mapDoble :: (a -> b -> c) -> [a] -> [b] -> [c]
mapDoble _ [] _ = []
mapDoble f (x:xs) (y:ys) = f x y : (mapDOble f xs ys)

mapDoble _ [] = const []
mapDoble f (x:xs) = \ys -> f x (head ys) : (mapDoble f xs (tail ys))

-------------
con foldr
mapDoble f xs = foldr (\x rec ys -> f x (head ys) : rec (tail ys)) (const [])
```

2.6 Ejercicio 8

I .

```
sumaMat :: [[Int]] -> [[Int]] -> [[Int]]
sumaMat = zipWith (\f1 f2 -> zipWith (+) f1 f2)
```

De manera más elegante :

```
sumaMat = zipWith (zipWith (+))
```

II . La ides es usar el llamado recursivo para que : $transponer(x : xs)$ = concatenar el iésimo elemento de x a la iésima lista de transponer xs. Si transponemos xs vacío se rompe. Separamos casos :

```haskell
separar :: [a] -> [[a]]
separar [] = []
separar (x:xs) = [x] : (separar xs)

transponer :: [[int]] -> [[int]]
transponer [] = []
transponer (x:xs) = if null (transponer xs) then separar x else zipWith (:) x (transponer xs)
```

Con folr

```haskell
separar = foldr (\x rec -> [x] : rec) []

transponer = foldr (\x rec -> if null rec then separar x else zipWith (:) x rec) []
```

3 Otras estructuras de datos

3.1 Ejercicio 9

Primero definimos un **foldNat** para foldear los naturales :

```haskell
foldNat :: b -> (Integer -> b) -> Integer -> b
foldNat cCero cSuc 0 = cCero
foldNat cCero cSuc n = cSuc (foldNat cCero cSuc (n-1))
```

Luego para la **potenciación** vemos cómo podemos definir a la potenciación de manera recursiva :

$$x^0 = 1$$

$$x^n = x \times x^{n-1}$$

O sea :

```haskell
potencia n 0 = 1
potencia n m = n * potencia n (m-1)
```

Que con foldNat es

```haskell
potenciacion n = foldNat 1 (n*)
```

3.2 Ejercicio 10

```haskell
genLista :: a -> (a -> a) -> Integer -> [a]
genLista e f 0 = []
genLista e f n = e : (genLista (f e) f (n-1))
```

Como es recursión estructural usamos **foldNat**

```haskell
genLista e f n = foldNat [] (\rec -> e : ???) n  ---- no sale
```

3.3 Ejericicio 11

Para el **foldPoli** seguimos la receta :

```haskell
data Polinomio a = X | Cte a | Suma (Polinomio a) (Polinomio a) | Prod (Polinomio a) (Polinomio a)
    deriving (Show)

foldPoli fX fCte fSuma fProd poli = case poli of
        X -> fX
        Cte a -> fCte a
        Suma i d -> fSuma (rec i) (rec d)
        Prod i d -> fProd (rec i) (rec d)
        where rec = foldPoli fX fCte fSuma fProd

evaluar :: Num a -> a -> Polinomio a -> a
evaluar a = foldPoli a id (+) (*)
```

Para la evaluación es simple, porque sabemos que para polinomios que tienen x, f(x) es a. En caso de constantes, identidad y luego la suma y el producto.

3.4 Ejercicio 12

```
data AB a = Nil | Bin (AB a) a (AB a)
foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
foldAB fNIl fBin (Bin i r d) = fBin (rec i) r (rec d)
        where rec = foldAB fNIl fBin

recAB :: b -> (AB a -> AB a -> b -> a -> b -> b) -> AB a -> b
recAB cNil cBin (Bin i r d) = cBin i d (rec i) r (rec d)
    where rec = recAB cNil cBin

esNil :: AB a -> Bool
esNil Nil = True
esNil _ = False

altura :: AB a -> Int
altura Nil = 0
altura (Bin i r d) = 1 + (max (altura i) (altura d))
---- con foldAB :
altura = foldAB 0 (\reci _ recd-> 1 + (max reci recd))

cantNodos  :: AB a -> Int
cantNodos Nil = 0
cantNodos (Bin i r d) = 1 + (cantNodos i) + (cantNodos d)
----- con foldAB :
cantNodos = foldAB 0 (\reci _ recd -> 1 + recIzq + recd)

mejorSegun p (Bin i r d)
        | esNil i && esNil d = r
        | esNil i = if p r (mejorSegun d) then r else mejorSegun d
        | esNil d = if p r (mejorSegun i) then r else mejorSegun i
        | otherwise = if p r (mejorSegun i) && (p r (mejorSegun d))  then r else if  p (mejorSegun i) (
            mejorSegun d)  then mejorSegun i  else mejorSegun d
```

// En el siguiente paso, f la pongo así porque en lambdas no puedo usar el formato de dividir por casos así foldAB no puedo usarlo porque tengo que acceder a i y a d para ver si son null. El tema es qué hacer en el caso base. La idea es dar un error, como en foldr1.

```
mejor Segun p = recAB (error   empty    A B   ) (f p)
        where f p i d recI r recD
        | esNil i && esNil d = r
                | esNil i = if p r recD then r else recD
                | esNil d = if p r recI then r else recI
                | otherwise = if p r recI && (p r recD)  then r else if  p recI recD then recI else recD

Usando la sugerencia:
mejor :: (a -> a -> Bool) -> a -> AB a -> a
mejor p r ab rec = if esNil ab then r  else (if p r rec then r else rec)

mejorSegun :: (a -> a -> Bool) -> AB a -> a
mejorSegun p (Bin i r d) =
    if p (mejor p r i (mejorSegun p i)) (mejor p r d (mejorSegun p d))
    then (mejor p r i (mejorSegun p i)) else (mejor p r d (mejorSegun p d)) mejor Segun p = recAB (error '
        empty AB' ) (f p)
        where f p recI r recD = if p (mejor p r i recI) (mejor p r d recD) then (mejor p r i recI) else
(mejor p r d recD)

esABB :: Ord a => AB a -> Bool                  esABB Nil = True
esABB (Bin i r d) = esABB i && (esABB d) &&
                        if esNil i then
                                if esNil d then True
                                else raiz d > r
                        else
                                if esNil d then raiz i <= r
                                else raiz i <= r && raiz d > r
raiz :: AB a -> a               raiz (Bin i r d) = r
esABB :: Ord a => AB a -> Bool
esABB Nil = True
esABB (Bin i r d)
        | esNil i && esNil d = True
    | esNil i = esABB d && raiz d > r
    | esNil d = esABB i && raiz i <= r
    | otherwise = esABB i && esABB d && raid d > r && ra z i <= r
Usando recAB (no puedo usar foldAB porque tengo que acceder a d y a i sin usarlos dentro de una llamada
    recursiva)
esABB ab = recAB True f
```

```
where f i d recI r recD
        | esNil i && esNil d = True
    | esNil i = recD && raiz d > r
    | esNil d = recI && raiz i <= r
    | otherwise = recI && recD && raid d > r && ra z i <= r
```

En esABB use recAB porque tenía que acceder al subárbol izquierdo y derecho sin usarlos como argumento de la recursión. En mejorSegun usé recursión primitiva porque debo poder acceder a los hijos izquierdo y derecho para poder preguntar si son null. En esNil, altura, y cantNodos usé foldAB porque 1) usa los parámetros del constructor del tipo AB SOLO para hacer llamados recursivos 2) 3) el caso base se escribe como una combinación de los parámetros del constructor base.

3.5 Ejericicio 13

```
ramas :: AB a -> [[a]]          ramas Nil = [[]]
ramas (Bin i r d) =
        | null (ramas i) && (null (ramas d)) = [[r]]
        | null (ramas i) = map (r:) (ramas d)
        | null (ramas d) = map (r:) (ramas i)
        | otherwise = map (r:) (ramas i ++ (ramas d))
```

– Si hacia directo map (r:) (ramas i ++ ramas) daria en el caso de Bin Nil r Nil la lista [[r],[r]] lo cual esta mal.

```
ramas ab = foldAB [[]] f
        where f recI r recD
                | null recI && (null recD) = [[r]]
                | null recI = map (r:) recD
                | null recD = map (r:) recI
                | otherwise = map (r:) (recI ++ recD)
cantHojas ab = foldAB 0 (\recI r recD -> if recI == 0 && (recD == 0) then 1 else recI+recD)
espejo ab = foldAB Nil (\recI r recD -> Bin recD r recI)


mismaEstructura Nil = esNil
mismaEstructrua (Bin i r d) = \ab -> case ab of
                        Nil -> False
                        Bin i2 r d2 -> mismaEstructura i i2 && (mismaEstructura d d2)
mismaEstructura = foldAB esNil (\recI r recD -> \ab -> case ab of
Nil -> False
Bin i2 r d2 -> recI i2 && recD d2)
```

Obs: En este caso, recI y recD son funciones que reciben un ab y dicen si este tiene la misma estructura que i o d respectivamente.

3.6 Ejercicio 14

```
data AIH a = Hoja a | Bin (AIH a) (AIH a)
// Todos los AIH tienen o dos hijos, o son hojas

foldAIH :: (a->b) -> (b -> b -> b) -> AIH a -> b
foldAIH fHoja fBin (Hoja x) = fHoja x
foldAIH fHoja fBin (Bin x y) = fBin (foldAIH fHoja fBin x) (foldAIH fHoja fBin y)

altura :: AIH a -> Integer
altura = foldAIH 1 (\recI recD -> max recI recD + 1)

tama o :: AIH a -> Integer
tama o = foldAIH 1 (+)
```

3.7 Ejercicio 15

```
data RoseTree a = Rose a [RoseTree a]

rose = Rose 2 [Rose 3 [], Rose 4 [Rose 5 []]]

foldRose :: (a -> [b] -> b) -> RoseTree a -> b
foldRose f (Rose r rs) = f r (map (foldRose f) rs)

hojas :: RoseTree a -> [a]
hojas (Rose x xs) = if null xs then [x] else concat (map hojas xs)

hojas = foldRT (\x rec -> if null rec then [x] else concat rec)
```

```
    xs es la lista que contiene todos los resultados de aplicarle hojas a cada sub-rosetree.

distancias = foldRose (\x rec -> if null rec then [0] else map (1+) concat rec)
rec es una lista de la funci n distancias aplicada a cada elemento de xs. si xs es vac o , rec va a ser
    vac o.

altura (Rose x xs) = 1 + if null (map altura xs) then 0 else maximum map altura xs
altura = foldRose (\x rec -> 1 + if null rec then 0 else maximum rec)
```

3.8 Ejercicio 16

```
data HashSet a = Hash (a -> Integer) (Integer -> [a])
vac o :: (a -> Integer) -> HashSet a
vac o f = Hash f (const [])

pertenece :: Eq a => a -> HashSet a -> Bool
pertenece e (Hash h table) = elem e table h e

agregar :: Eq a => a -> HashSet a -> HashSet a
agregar e (Hash h table) = Hash h (\n -> if n == (h e) && not elem e table n
then (e:table n) else table n)

interseccion :: Eq a => HashSet a -> HashSet a -> HashSet a
interseccion (Hash h1 table1) (Hash h2 table2) = Hash h1 table3
        where table3 = \n -> filter (\x -> pertenece x (Hash h2 table2)) (table1 n)

data Lista a = [] | a : Lista a
foldr1 :: (a -> b -> b) -> [a] -> b
    No recibe nada para el caso base ya que como caso base se devuelve al unico elemento de la lista.
Esto hace que el tipo de salida sea el mismo que el tipo de la lista.

foldr1 :: (a -> a -> a) -> [a] -> a
foldr1 f [x] = x
foldr1 f (x:xs) = f x (foldr1 f xs)

foldr1 f [] = error   empty   l i s t
foldr1 f (x:xs) = recr (\ x   xs rec -> if null xs then  x   else f x rec) x (x:xs)
Usamos recr para poder acceder a xs y ver si ya estoy en el caso del caso base
```

4 Generación Infinita

4.1 Ejercicio 17

```
[ x | x <- [1..3], y <- [x..3], (x + y) `mod` 3 == 0 ]
```

- $x \mid x < -[1..3]$: x toma los valores del 1 al 3.
- $y < -[x..3]$: Para cada valor de x, y toma los valores desde x hasta 3 (es decir, y depende de x).
- $(x + y) \mod 3 == 0$: Sólo se incluyen las combinaciones de (x, y) en las que la suma de x e y sea divisible por 3.
- $[x \mid ...]$: Finalmente, la lista resultante contiene el valor de x para cada combinación que cumpla las condiciones anteriores. Es decir, $[1, 3]$ ya que $x = 1 + y = 2$ y $x = 3 + y = 0$

4.2 Ejercicio 18

```
paresDeNat :: [(Int, Int)]
paresDeNat = [p | k <- [0..], p <- paresQueSuman k]
paresQueSuman :: Int -> [(Int, Int)]
paresQueSuman k = [(x,k-x) | x <- [0..k]]
```