Práctica 1: Programación Funcional

Tomás Felipe Melli

May 12, 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
```

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
```

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 nos numeros son de tipo Float, tenemos que modificar la clase de tipo a **Fractional**

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

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

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

Y esta es un ejemplo de aplicar parcialmente la función *substract* dejando en evidencia que *substract* **está currificada** :

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

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
```

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.

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, simplemente podríamos sumar 0.

  ```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
  ```

  ```haskell
  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.

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

```
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.

```
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 ...

```
sumasParciales :: (Num a) => [a] -> [a]
sumasParciales xs = foldl (\y x -> if null y then [x] else y ++ [last y + x]) [] xs
```

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 :

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

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 :

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

2.1.5 V

Lo mismo pero

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

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]
```

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)
```

Recordemos de la T1 :

Sea `g :: [a] -> b` definida por dos ecuaciones:

$$\begin{array}{lcl} \texttt{g []} & = & \langle \textit{caso base} \rangle \\ \texttt{g (x : xs)} & = & \langle \textit{caso recursivo} \rangle \end{array}$$

g está dada por **recursión estructural** si:

1. El caso base devuelve un valor fijo z.
2. El caso recursivo se escribe usando (cero, una o muchas veces) x y (g xs), pero sin usar el valor de xs ni otros llamados recursivos.

$$\begin{array}{lcl} \texttt{g []} & = & \texttt{z} \\ \texttt{g (x : xs)} & = & \ldots \texttt{x} \ldots (\texttt{g xs}) \ldots \end{array}$$

Si miramos la línea

```
elementosEnPosicionesPares :: [a] -> [a]
...
        else x : elementosEnPosicionesPares (tail xs)
```

La función **elementosEnPosicionesPares** devuelve una lista (como su nombre declarativo indica) con los elementos en las posiciones pares. Para ello hace recursión sobre la cola de xs, es un detalle no menor, pero en la definición, *el caso recursivo se escribe usando x y (g xs), sin usar el valor de xs ni otros llamados recursivos*, nos da a entender que nosotros no podemos desentendernos de la estructura original que nos pasan. Consultar esto igual. Por lo que entiendo, no podemos hacer recursión sobre una estructura que no sea xs (ejemplo: g sacarMínimo(xs)). Con respecto al caso base, no hay problema que devuelva la lista vacía (chequear esto también)
La función **entrelazar** básicamente toma un elemento de la lista a y otro de la lista b, y así

$$entrelazar[1,2,3][4,5,6]$$
$$= [1,4,2,5,3,6]$$

En este caso, la función hace la recursión sobre xs

```
    then x : entrelazar xs []
    else x : head ys : entrelazar xs (tail ys)
```

Y siguiendo la idea de que no se modifica la estructura original, vale que se mantiene. El caso base retorna un z digamos. Por tanto estaría bien en decir que se trata de una **recursión estructural**. Vamos a definirla usando foldr (**Toda recursión estructural es una instancia de foldr.**).

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

Veamos un esquema de cómo funciona esto para un ejemplo particular :

$$sacarUna\ 1\ [1,2,1]$$

Hace la llamada a λ :

$$= \lambda\ 1\ [2,3](recr\ (x\ xs\ rec \rightarrow if\ x == 1\ then\ xs\ else\ x : rec)[][2,1])[$$

Evalúa entonces la primer guarda y devuelve **xs**

$$= [1,2]$$

No me queda claro qué pasa con la otra llamada (preguntar).

2.4.2 Por qué no sirve foldr en este caso ?

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

sacarUna' a = foldr (\x rec -> if x == a then rec else x : rec) []
----------------------
ghci> sacarUna' 1 [1,2,3,4,1]
[2,3,4]
ghci> sacarUna 1 [2,3,4,1,1]
[2,3,4,1]
```

La explicación está en que con recr, *si x == a devolvemos xs*, y con foldr continúa la recursión hasta cubrir toda la lista.

2.4.3 insertarOrdenado

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

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

Es el análogo a **zip**

```
ghci> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

2.6 Ejercicio 8

Resuelto :

```
sumaMat :: [[Int]] -> [[Int]] -> [[Int]]
sumaMat = foldr (\xs rec yss -> zipWith (+) xs (head yss) : rec (tail yss)) (const [])

trasponer :: [[Int]] -> [[Int]]
trasponer (xs : xss) = foldr (zipWith (:)) [[] | i <- [1 .. (length xs)]] xss
```

3 Otras estructuras de datos

3.1 Ejercicio 9

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

```
foldNat :: (Int -> b -> b) -> b -> Int -> b
foldNat _ z 0 = z
foldNat f z n = f n (foldNat f z (n-1))
```

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

$$x^0 = 1$$

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

Entonces :

```
potenciacion n = foldNat (\x rec -> n * rec) 1
---------------------
ghci> potenciacion 2 3
8
```

3.2 Ejercicio 10

Resuelto :

```
genLista :: a -> (a -> a) -> Integer -> [a]
genLista startElement incrementFunction size = foldl (\acc x -> if null acc then [startElement] else acc
    ++ [incrementFunction (last acc)]) [] [1 .. size]

desdeHasta :: Integer -> Integer -> [Integer]
desdeHasta inicio fin = genLista inicio (+ 1) (fin - inicio) ++ [fin]
```

3.3 Ejericicio 11

Para el **foldPoli** seguimos la receta :

```
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 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

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

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

altura = foldAB (const 0) (\reci _ recd-> 1 + max reci recd)

cantNodos = foldAB (const 0) (\reci _ recd -> 1 + recIzq + recd)
```

3.5 Ejericicio 13

Resuelto:

```haskell
ramas :: AB a -> [[a]]
ramas = foldAB [] (\recIzq r recDer -> if null recIzq && null recDer then [[r]] else map (r :) (recIzq ++ recDer))

mismaEstructura :: AB a -> AB b -> Bool
mismaEstructura = foldAB esNil (\recIzq r recDer arbol -> not (esNil arbol) && recIzq (hijoIzquierdo arbol) && recDer (hijoDerecho arbol))

hijoIzquierdo :: AB a -> AB a
hijoIzquierdo (Bin izq _ _) = izq

hijoDerecho :: AB a -> AB a
hijoDerecho (Bin _ _ der) = der
```

3.6 Ejercicio 15

```haskell
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)

hojasRose :: RoseTree a -> [a]
hojasRose = foldRose (\r rec -> if null rec
                                then [r]
                                else concat rec)

ramasRose :: RoseTree a -> [[a]]
ramasRose = foldRose (\r rec -> if null rec
                                then [[r]]
                                else map (r:) (concat rec))

tama oRose :: RoseTree a -> Int
tama oRose = foldRose (\_ rec -> 1 + sum rec)

distanciasRose :: Rosetree a -> [(a, Int)]
distanciasRose = foldRose (\r rec -> if null rec
                                     then [(r, 0)]
                                     else map (\(tree, distance) -> (tree, distance + 1)) (concat rec))

alturaRose :: RoseTree a -> Int
alturaRose = foldRose (\_ rec -> if null rec
                                 then 1
                                 else 1 + maximum rec)
```

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 = [(b, c) | a <- [0 ..], b <- [0 .. a], c <- [0 .. a], b + c == a]
```