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.: $data\ Color\ =\ Rojo \mid Verde \mid Azul$. 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 $a \to b$ Ej.:

```
add :: Int -> Int -> Int
add x y = x + y

add5 :: Int -> Int
add5 = add 5

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.

```
add5 3 -> Este 3 llega como "y" a add

add5 = add 5
add = 5 + y
add = 5 + 3
add = 8
```

Otro ejemplo

```
const :: a -> b -> a
const x y = x

const (const 1) 2 -> const 1
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 ídea seria 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 $map(\backslash x \to x\ 0)\ (map(+)\ [1,2,3])$. En este caso la asociatividad a la izquierda nos muestra algo así $map(\backslash x \to x\ 0)\ ((map(+)\ [1,2,3]))$
Esto genera $map(\backslash x \to x\ 0)\ [2,3,4]$ 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 -> b) -> (a -> 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 $(:)\ ::\ a -> [a] -> [a]$.
- 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 quisieramos 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 quisieramos 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 $a \to (b \to (c \to (d)))$ 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.

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

- $fst :: (a, b) \to a$
- $cargarStringArray :: String- > [] \to [String]$. 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 $(Numa, Orda)$.
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: $func :: Num\ a \implies a \to a \to a$
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.

- $sumarUno :: a- > a$: No falla nunca.
- $division :: a- > b$: 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 comodin: _.
- 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 cómputos o cálculo de los valores.

¿Cómo evalúamos 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.

¿Cuando se detiene la evaluación de una expresión?

- Cuando el programa estalla.
  - Loop infinito.
  - Indefinición.
- Cuando la expresión es una función **parcialmente aplicada**. ¿que seria 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.

Cambiemos 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 $(18 \leq x)$

Currificación

Una función es currificada cuando tiene la siguiente forma $ab :: Int- > Int- > Int$
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 $ab :: Int- > (Int- > Int)$
La función Curry **NO CAMBIA** la manera en que la función original currificada recibe los argumentos, sino que, la función curry es una especie de *puente* para que nosotros mandemos los argumentos separados y los aplique a la función no

currificada.
Importante: No tiene sentido hablar de currificación 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 $suma :: (Int, Int) -> Int -> Int$ 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 $(g \ . \ f)$
Recordemos que en Álgebra vimos Composición de Funciones y es algo así: sean $A : B \to C$ y $D : A \to B$

La composición gof es $A \to B$. 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

```
ent: Entrada sal: Salida

(.) :: (b -> c) -> (a -> b) -> (a -> c)

Queremos: (g . f) x ⇒ g (f x)

Desglosemos
(.) :: (ent. de g -> sal. de g) -> (ent. de f -> sal. de f) -> (ent. de f -> sal. de g)

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.

Entonces, primero se evalúa f x, lo cual produce un resultado de tipo b.

Luego, este resultado (de tipo b) se pasa a g, produciendo un resultado de tipo c.

(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.

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

Recordemos que esto es algo recordable pues fog seria 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

$not(nullx) \equiv (not.null)x$
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.: $(== 0).modnm$ no funciona, pero $(== 0).modn$ sí.

Funciones Lambda

Son funciones anónimas, es decir, no tienen nombre.
$(\backslash x \to x + 1)$: Es una función anónima que dado un x, te devuelve una función que le aplica x+1.

Funciones Lambda Anidadas

$\backslash y \to (\backslash x \to y)$: 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 $\backslash y \to \backslash x \to y$.
Mejor notación para funciones anidadas: $\backslash y\ x \to y$

Reducción de Lambda

Preguntar $foldr(\backslash x\ rec \to fx : rec) \equiv \backslash x \to (:)(fx)$ y $\backslash x \to ex \equiv e$

Asignar nombre a una función

$a = \backslash x \to x$: al nombre **a** le asigno la función anónima $\backslash x \to x$

¿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
```

```
                then x : pares xs
                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**.

```
g [] = []
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.

```
filter :: (a -> Bool) -> [a] -> [b]
filter p [] = []
filter p (x : xs) = if p x
                    then x : filter p xs
                    else filter p xs
```

La función que le envíamos 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.

```
eliminarImpares :: [a] -> [b]
eliminarImpares xs = filter (\x -> x `mod` 2 == 0) xs

borrarNegativos :: [a] -> [b]
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

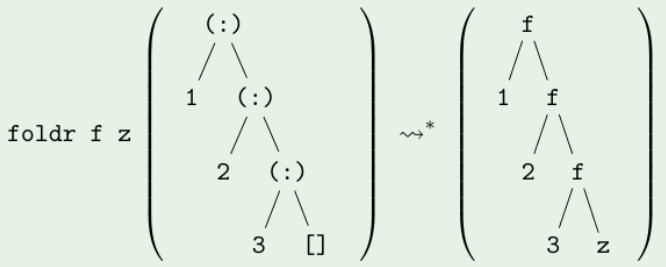
¿Qué tienen en común los siguientes problemas? ¿En qué difieren?

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

reverso :: [a] -> [a]
reverso [] = []
reverso (x:xs) = reverso xs ++ [x]

sum :: [int] -> int
sum [] = 0
sum (x:xs) = x + sum xs

En común:
    1. Todos tienen un caso base, pero es diferente.
    2. Todos hacen un paso recursivo, pero cambia la operación que hacen.
    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:




```
suma :: [Int] -> Int
suma = foldr (+) 0

 suma [1, 2]  ⇝   foldr (+) 0 [1, 2]
              ⇝   (+) 1 (foldr (+) 0 [2])
              ⇝   (+) 1 ((+) 2 (foldr (+) 0 []))
              ⇝   (+) 1 ((+) 2 0)
              ⇝*  3
```

- 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

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

Por lo tanto foldr se define como

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

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

- 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? $foldr(\backslash x\ r \to x)\ 0\ [1,2]$

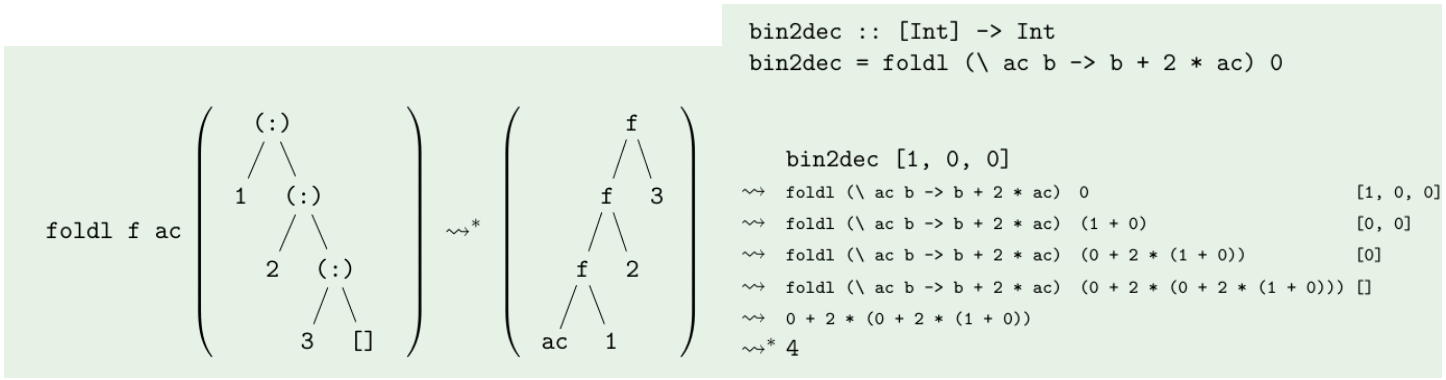
- Toma la lista y hace recursión. Empieza con el 2, ejecuta la función y devuelve 2. Hace el paso recursivo.
- Toma la lista y hace recursión. Ahora sigue con el 1, recibe 1 y la salida es 1.
- Esta función agarra el primer elemento de una lista, sería el head.

IMPORTANTÍSIMO: Se pueden recorrer dos listas, tres listas, las que quieras a la vez con Foldr. La recursion se hace sobre una sola, pero las n-1 listas las enviarías por argumento en cada paso recursivo. Véase **anexo** para ver ver el árbol de recursión y un ejemplo práctico. **Importante**: Foldr puede trabajar con listas infinitas.
Véase **anexo** para ver ejemplos de Foldr.

Recursión que no es estructural

```
ssort :: Ord a => [a] -> [a]
ssort [] = []
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

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

¿Qué es lo que sucede en el siguiente ejemplo?

```
foldl (\x y -> 1) 0 unos
foldl f (f 0 1) unos
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 $Foldable\ t \implies (b \to a \to b) \to b \to ta \to b$

- 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

```
g [] = b
g (x:xs) = f x xs (g xs)
```

Por lo tanto recr se define como

```
recr :: (a -> [a] -> b ->b) -> b -> [a] -> b
recr f z [] = z
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: $dataDia = Lu|Ma|Mi|Mie|Ju|Vi|Sa|Do$
Lu, Ma, Mi, ... son constructores del tipo dia.
Los argumentos podrán ser recibidos diciendo qué tipo se espera, y usamos pattern matching para utilizarlos.

```
EsLunes :: Dia -> Bool
EsLunes Lu = True
EsLunes _ = False

EsFinDeSemana :: Dia -> Bool
EsFinDeSemana Sa = True
EsFinDeSemana Do = True
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

```
Edad :: Persona -> Int
Edad (LaPersona n a e) = e

Cumpleaños :: Persona -> Persona
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 — Succ Nat

Tipos Polimórficos

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

```
data List a = Vacia | Const a (List a)
```

Importante en Tipos

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

```
data List = Vacia | Cons Int ListI
data List a = Vacia | Cost a (List a)

Error. No puede estar Vacia 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 — selectores, condiciones]. Ej.: [(x,y) — x ¡- [0..5], y ¡- [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 decirque 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: $Foldable\ t \implies (a \to b \to b) \to b \to t\ a \to b$
Donde b era el caso base / recursivo y a el elemento actual, acá tenemos dos.
Recordemos la estructura del tipo AB: AB a = Nil — 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 → a → b → b) → b → AB a → b donde :t Bin: (AB a) → a → (AB) a → AB a donde

- $(b \to a \to b \to b)$
  - 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

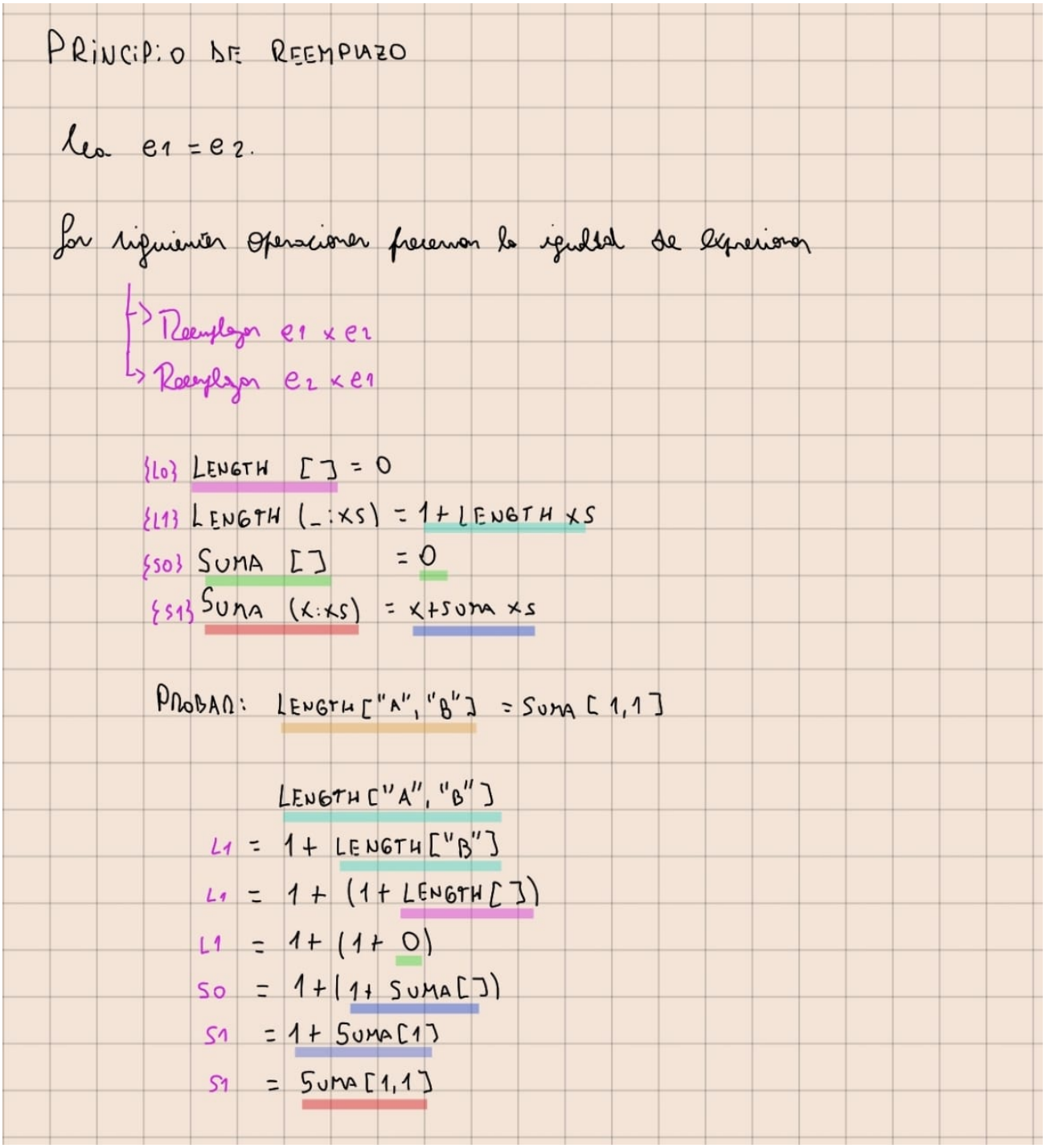
- Trabajaremos con estructuras de datos **finitas**. Más técnicamente, con tipos de datos **inductivos**.
- Trabajamos con **funciones totales**
  - Las ecuaciones deben cubrir todos los casos posibles.
  - La recursión siempre termina.
- El programa **no depende del orden** de las ecuaciones.

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**.

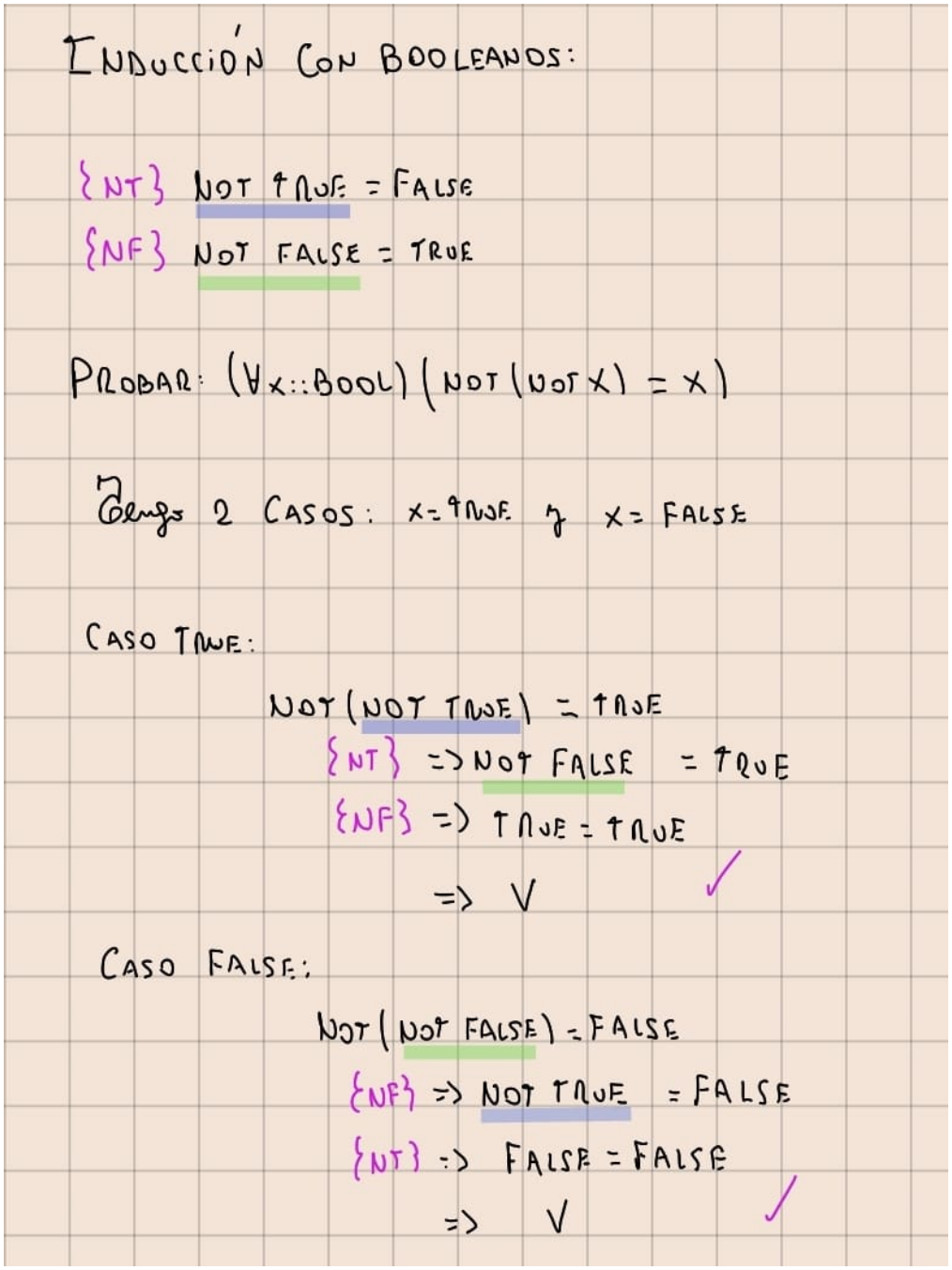


Inducción Estructural

Cada tipo de datos tiene su propio principio de inducción.
Importante: El . acá cumple el rol del () en el ∀ es decir $\forall x :: Bool.\mathcal{P} \equiv (\forall x :: Bool)(\mathcal{P})$ y el :: cumple el rol de **es del tipo...**

Inducción sobre booleanos

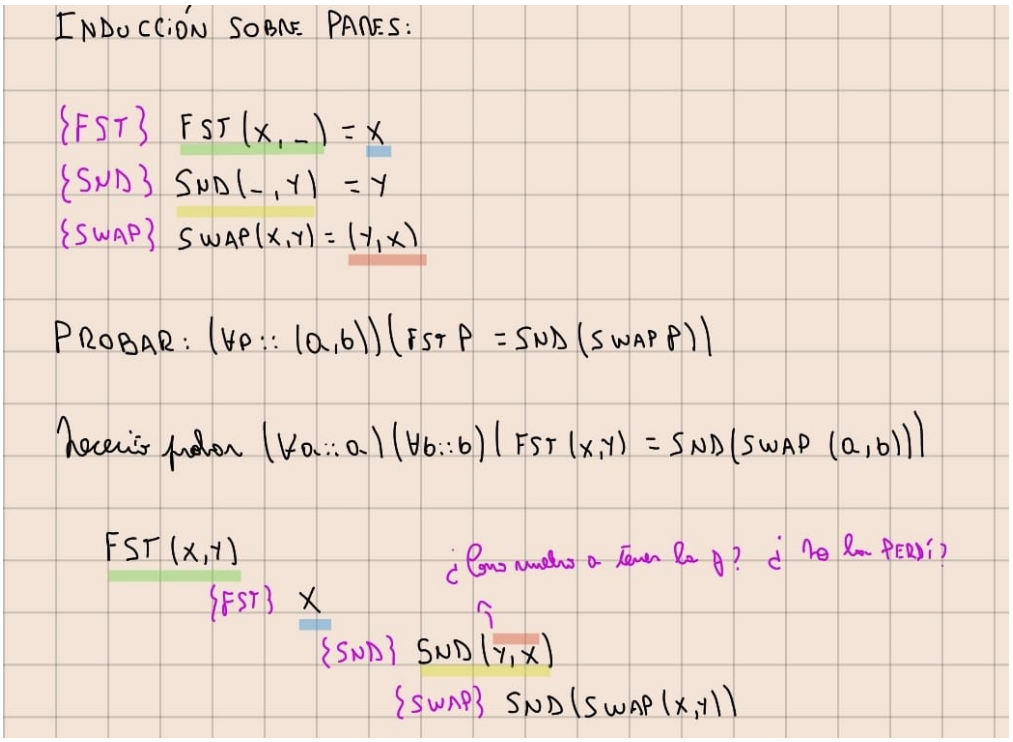
Si $\mathcal{P}(True)$ y $\mathcal{P}(False)$ entonces $\forall x :: Bool\ \mathcal{P}(x)$



Inducción sobre pares

Si $(\forall x :: a)(\forall y :: b)(\mathcal{P}(x, y))$ entonces $\forall p :: (a, b)\mathcal{P}(p)$

```
{FST}  fst (x, _)  = x
{SND}  snd (_, y)  = y
{SWAP} swap (x, y) = (y, x)
Para probar ∀p :: (a, b). fst p = snd (swap p)
basta probar:
∀x :: a. ∀y :: b. fst (x, y) = snd (swap (x, y))

fst (x, y) = x = snd (y, x) = snd (swap (x, y))
             ↑   ↑           ↑
            FST  SND        SWAP
```



Inducción sobre naturales

Si $\mathcal{P}(Zero)$ y $(\forall n :: Nat)(\mathcal{P}(n) \implies \mathcal{P}(Suc\ n))$ entonces $(\forall n :: Nat)(\mathcal{P}(n))$ donde

- $\mathcal{P}(n)$: Hipótesis Inductiva.
- $\mathcal{P}(Suc\ n)$: Tesis Inductiva.

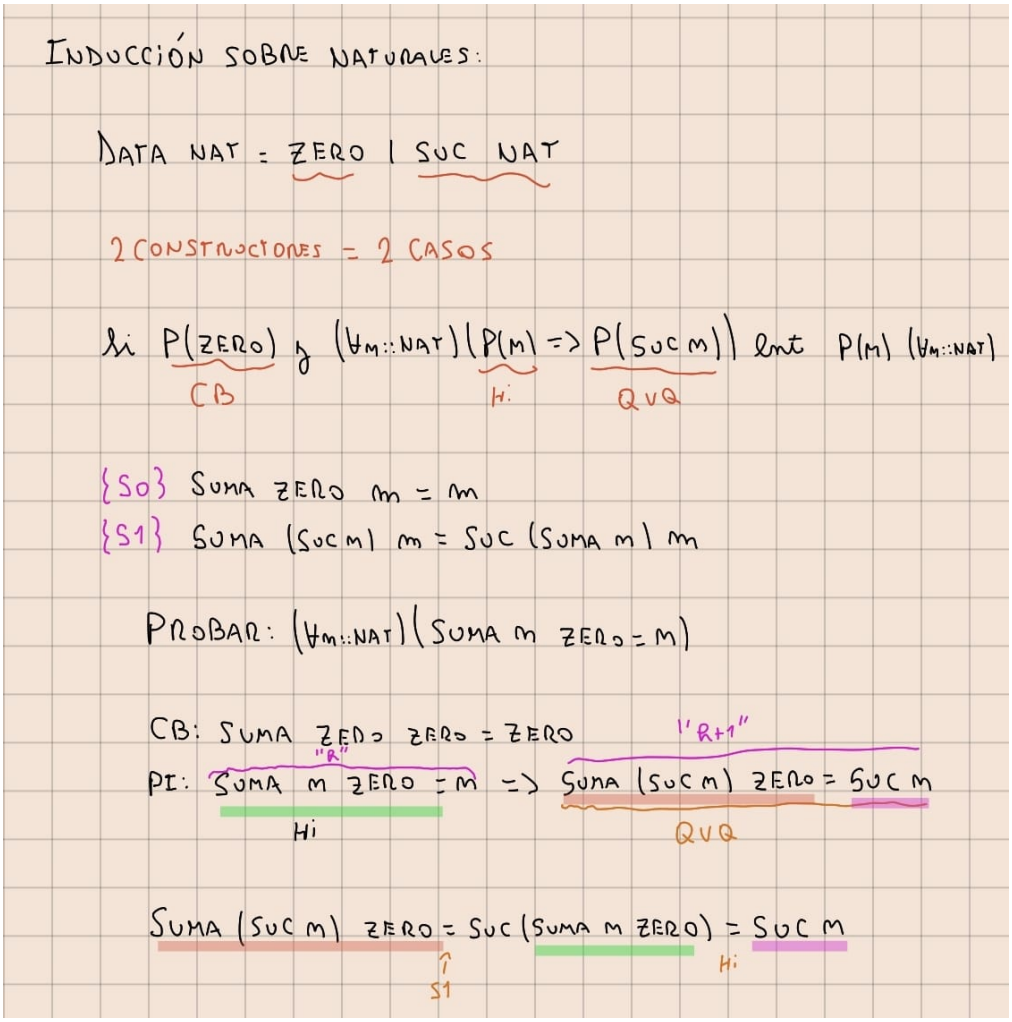
Ejemplo

{S0} suma Zero m = m
{S1} suma (Suc n) m = Suc (suma n) m

Para probar ∀n :: Nat. suma n Zero = n
basta probar:

1. suma Zero Zero = Zero.
   Inmediato por S0.
2. suma n Zero = n ⇒ suma (Suc n) Zero = Suc n.
   (suma n Zero = n: H.I. — suma (Suc n) Zero = Suc n: T.I.)

suma (Suc n) Zero = Suc (suma n Zero) = Suc n
(= ↑ S1; = ↑ H.I.)



Inducción Estructural: Caso General

Sea $\mathcal{P}$ una propiedad acerca de las expresiones tipo T tal que

- $\mathcal{P}$ vale sobre todos los constructores base de T,
- $\mathcal{P}$ vale sobre todos los constructores recursivos de T, asumiendo como hipótesis inductiva que vale para los parámetros de tipo T,

entonces $(\forall x :: T)(\mathcal{P}(x))$

Principio de Inducción sobre Listas

INDUCCIÓN SOBRE LISTAS:

DATA $[a] = [] \mid a : [a]$ 2 CONSTRUCCIONES = 2 CASOS

si $P([])$ y $(\forall x :: a)(\forall xs :: [a])(P(xs) \Rightarrow P(x:xs))$ ent $(\forall xs :: [a])(P(xs))$

CB Hi QvQ

{M0} MAP f [] = []
{M1} MAP f (x:xs) = f x : MAP f xs
{A0} [] ++ ys = ys
{A1} (x:xs) ++ ys = x : (xs ++ ys)

CB: $P([])$
PI: $(\forall x :: a)(\forall xs :: [a])(P(xs) \Rightarrow P(x:xs))$

PROPIEDAD: si $f :: a \to b$, $xs :: [a]$, $ys :: [a]$ ent

P = MAP f (xs ++ ys) = MAP f xs ++ MAP f ys
Hi = P(xs): MAP f (xs++ys) = MAP f xs ++ MAP f ys
QvQ = P(x:xs): MAP f ((x:xs)++ys) = MAP f (x:xs) ++ MAP f ys

P([]) = xs = []

CB: MAP f ([] ++ ys)
{A0} MAP f ys ⇝ (FORMA: [] ++ ys DONDE ys = MAP f ys)
{A0} [] ++ MAP f ys
{M0} MAP f [] ++ MAP f ys
se cumple P([]) ⇒ MAP f xs ++ MAP f ys para xs = []

Pi: MAP f ((x:xs) ++ ys)

{A1} MAP f (x : (xs ++ ys))
{M1} f x : MAP f (xs++ys)
{Hi} f x : MAP f xs ++ MAP f ys (FORMA: x : (xs ++ ys))
(x, xs, ys)
{A1} (f x : MAP f xs) ++ MAP f ys
{M1} MAP f (x:xs) ++ MAP f ys
QvQ

Principio de Inducción sobre Árboles Binarios

Ejemplo: principio de inducción sobre árboles binarios

```
data AB a = Nil | Bin (AB a) a (AB a)
```

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `AB a` tal que:

- $\mathcal{P}(\texttt{Nil})$
- $\forall \texttt{i} :: \texttt{AB a}.\ \forall \texttt{r} :: \texttt{a}.\ \forall \texttt{d} :: \texttt{AB a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{i}) \wedge \mathcal{P}(\texttt{d}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Bin i r d})}_{\text{T.I.}})$$

Entonces $\forall \texttt{x} :: \texttt{AB a}.\ \mathcal{P}(\texttt{x})$.

Principio de Inducción sobre Polinomios

Ejemplo: principio de inducción sobre polinomios

```
data Poli a = X
            | Cte a
            | Suma (Poli a) (Poli a)
            | Prod (Poli a) (Poli a)
```

Sea $\mathcal{P}$ una propiedad sobre expresiones de tipo `Poli a` tal que:

- $\mathcal{P}(\texttt{X})$
- $\forall \texttt{k} :: \texttt{a}.\ \mathcal{P}(\texttt{Cte k})$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Suma p q})}_{\text{T.I.}})$$
- $\forall \texttt{p} :: \texttt{Poli a}.\ \forall \texttt{q} :: \texttt{Poli a}.$
  $$(\underbrace{(\mathcal{P}(\texttt{p}) \wedge \mathcal{P}(\texttt{q}))}_{\text{H.I.}} \Rightarrow \underbrace{\mathcal{P}(\texttt{Prod p q})}_{\text{T.I.}})$$

Entonces $\forall \texttt{x} :: \texttt{Poli a}.\ \mathcal{P}(\texttt{x})$.

Relación entre foldr y foldl

Propiedad: Si $f :: a \to b \to b,\ z :: b,\ xs :: [a]$ entonces foldr f z xs = foldl (flip f) z (reverse xs)
Lema: Si $g :: b \to a \to b,\ z :: b, x :: a, xs :: [a]$ entonces foldl g z (xs ++ [x]) = g (foldl g z xs) xs

Puntos de vista intensional vs extensional

Sí, es con **s**. Es intensional.
¿Es equivalente mergesort = insertionSort? La realidad es que: hacen lo mismo (llegan al mismo resultado); ordenan algo pero de una manera diferente.

- Punto de vista intensional: Dos valores son iguales si están definidos de la misma manera.
- Punto de vista extensional: Dos valores son iguales si son indistinguibles al observarlos.

Entonces mergesort = insertionSort desde el lado extensional, porque hacen lo mismo pero no son iguales desde el punto de vista intensional.

Principio de Extensionalidad Funcional

Sean $f,\ g :: a \to b$.
Principio de extensionalidad funcional: Si $(\forall x :: a)(f\ x = g\ x)$ entonces $f = g$
El ppio. de extensionalidad funcional nos dice que si son funciones son iguales entonces son iguales punto a punto donde las evaluemos.

Demostración de Desigualdades

¿Cómo demostramos que no vale una igualdad e1 = e2 :: A?
Entiendo que básicamente hay que encontrar hacer **ALGO** para demostrar que no son iguales. Este **ALGO** es una funcionalidad que devuelva algo con uno, y otra cosa con otro. Por lo tanto debería ser del tipo $obs\ a :: A \to Bool$.
Demostrar que **no** vale la igualdad: $id = swap :: (Int, Int) \to (Int, Int)$.
Esto es fácil de probar porque sabemos que id nos da exactamente lo mismo que envíamos, y swap justamente da vuelta todo. Por lo tanto podemos comparar el primer elemento con id y el primer elemento haciendo el swap.
Ej.: $(1, 2) \to id(1, 2) \to (1, 2)$ pero $swap(1, 2) = (2, 1)$ y el primer elemento de $id(1, 2) \to (1, 2)$ es decir, $1 \neq 2$ que arroja el swap.
Por lo tanto

```
Ej: (1, 2)
obs :: ((Int, Int) -> (Int, Int)) -> Bool
obs f = fst (f (1,2)) == 1

obs id -> True
obs swap False
```

Nótese que el obs está **exactamente armado para este caso particular**. Para demostrar que las funciones no hacen lo mismo.

Isomorfismo de Tipos

Decimos que dos tipos son isomorfos si podemos pasar de uno al otro aplicando alguna función y al componerlas el resultado arroja la función identidad.

```
("hola", (1, True)) :: (String, (Int, Bool))
((True, "hola"), 1) :: ((Bool, String), Int)
Estos dos tipos son isomorfos porque podemos transformar los valores de un tipo en valores del otro

f :: (String, (Int, Bool)) -> ((Bool, String), Int)
f (s, (i, b)) = f((b, s), i)

g :: ((Bool, String), Int) -> (String, (Int, Bool))
g ((b, s), i) = f (s, (i, b))
```

¡Es básicamente crear una función que reciba los parámetros de otra manera y los mande a la otra función de la otra forma! Una especie de Curry/Uncurry.
Con esto se puede demostrar que $g.f = id$ y $f.g = id$
Formalmente, podemos decir que dos tipos de datos A y B son **isomorfos** si

- Hay una función $f :: A \to B$ total.
- Hay una función $g :: B \to A$ total
- Se puede demostrar que $g.f = id :: A \to A$
- Se puede demostrar que $f.g = id :: B \to B$

En criollo: Debe existir una función que me mapee de f a g y viceversa. Por último, si hago f(g) o g(f) tienen que dar la identidad.
Notación: $A \simeq B$ indican que A y B son isomorfos.

Intérpretes

Un intérprete ejecuta programas, no los traduce como si fuese un compilador.

Sintaxis Concreta

Se pueden representar programas como cadenas de texto. Es tedioso de interpretar algo así: if $x > 0$ then x else -x

Sintaxis Abstracta

Podemos representar instrucciones a través de árboles: (Eif (gt (var "x") cte 0)) (var("x")) (neg (var("x")))

Lenguaje de Expresiones Aritméticas

Veamos como hacer un intérprete que pueda reconozca números y haga sumas.
Primero tenemos que definir los tipos de expresiones que vamos a querer reconocer (expresiones) en nuestro intérprete.

```
data Expr = EConstNum Int
          | EAdd Expr Expr
```

Nótese que EAdd es un constructor del tipo Expr recursivo que en algún momento llegará al caso base de EConstNum que devuelve un Int.
¿Es posible comparar los resultados de las Expr desde una manera de igualdad? ¡No! Los tipos no aceptan **ningún tipo de operación** a menos que lo indiquemos. Si queremos que estos tipos puedan compararse de la siguiente forma $n_1 = n_2$ tenemos que configurar **deriving (EQ)** donde EQ es una clase.

```
data Expr = EConstNum Int
          | EAdd Expr Expr
          deriving Eq
```

¿Es posible mostrar el resultado en consola de las operaciones que hagamos? ¡No! Los tipos no aceptan **ningún tipo de operación** a menos que lo indiquemos. Si queremos que el resultado pueda mostrarse en consola de alguna manera específica tenemos que hacer una instancia de Show.

```
instance Show Expr where
show = showExpr
  where
    showExpr :: Expr -> String
    showExpr (EConstNum n) = show n
    showExpr (EAdd e1 e2) = "(" ++ show e1 ++ "+" ++ show e2 ++ ")"
```

Bueno, ahora sí. ¿Cómo podemos usar todo esto para computar nuestras expresiones? Necesitamos de alguna manera una función que nos permita enviar expresiones, tomarlas por pattern matching y hacer algo.

```
evalExpr :: Expr -> Int
evalExpr (EConstNum n) = n
evalExpr (EAdd n m) = evalExpr n + evalExpr m
```

Pero ¿cómo lo usamos?. Primero tenemos que llamar a la función enviando la información que nos pide, es decir, algo del tipo Expr.
Probemos hacer 1+2, esto en sintáxis concreta sería algo como: $evalExpr\ (EAdd\ (EConstNum\ 1)\ (EConstNum\ 2))$. ¡Nótese que estamos utilizando los constructores del tipo!
Entonces, el resultado es: $ghci >\ evalExpr\ (EAdd\ (EConstNum\ 1)\ (EConstNum\ 2)) \implies 3$

Nuestro Lenguaje de Expresiones Aritméticas con Booleanos

Queremos agregarle booleanos a nuestro lenguaje. ¿Qué debemos hacer? Lo primero es lo primero, tenemos que ver si debemos refactorizar algo o simplemente extender. Como hasta ahora hicimos solamente cosas de números raramente vamos a tener que refactorizar algo aunque eso está por verse.
Por lo tanto lo primero que debemos hacer es aceptar que el booleano sea un constructor del tipo Expr

```
data Expr =   EConstNum Int
            | EConstBool Bool
            | EAdd Expr Expr
```

Entonces ahora podemos extender la clase Show

```
    ... mismo antes
    showExpr (EConstBool b) = show b
```

Por último, podemos aceptar evaluar la expresión de booleano (que sería un caso base) ante eventuales operaciones de este tipo.

```
evalExpr :: Expr -> Int
evalExpr (EConstNum n) = n
evalExpr (EConstBool b) = b
evalExpr (EAdd n m) = evalExpr n + evalExpr m
```

¿Cuál es el problema que estamos viendo? El problema es que ahora evalExpr **no** solo devuelve Int sino que Bool. Por esto, podemos crear un nuevo tipo que sean los tipos de salida de nuestras expresiones.

```
data Val = VN Int | VB Bool  deriving Eq
```

Luego,

```
evalExpr :: Expr -> Val
evalExpr (EConstNum n) = n
evalExpr (EConstBool b) = b
evalExpr (EAdd n m) = evalExpr n + evalExpr m
```

Pero ahora hay otro problema: **evalExpr n y evalExpr m devuelven un tipo Val, el tipo Val es un Int — Bool**. Veamos para qué tipos está permitida la suma infija (Num a =¿a -¿a -¿a). Por lo tanto, **ahora que estamos usando Val no podemos usar la suma porque Val no necesariamente es un número, también puede ser un booleano**. Así que como nosotros sabemos que **vamos a usar el +** cuando tenemos números hay que hacer una función específica para ese tipo.
Creamos entonces, una función que vamos a llamar para que nos devuelva una nueva instancia del tipo Val que haga las sumas correspondientes

```
addVal :: Val -> Val -> Val
addVal (VN n) (VN m) = VN (n+m)
addVal _ _ = error "Algún sumando no es numérico"
```

Por lo tanto, reemplazamos + con addVal

```
evalExpr :: Expr -> Val
evalExpr (EConstNum n)  = VN n
evalExpr (EConstBool b) = VB b
evalExpr (EAdd n m) = addVal (evalExpr n) (evalExpr m)
```

¿Qué falta ahora? Configurar el show de nuestra nueva clase Val porque evalExpr nos devuelve algo de tipo Val y todavía no definimos cómo mostrarlo.

```
instance Show Val where
show = showVal
    where
        showVal :: Val -> String
        showVal (VN n) = show n
        showVal (VB b) = show b
```

Por último, ejecutamos igual que antes y funcionará todo: $evalExpr\ (EAdd\ (EConstNum\ 1)\ (EConstNum\ 2)) \rightarrow 3$

Nuestro Lenguaje de Expresiones con Definiciones en el Ambiente

Queremos aceptar expresiones del tipo $let\ x = 3\ in\ (let\ y = x + x\ in\ 1 + y)$
Recordemos que en Haskell, un let define una variable en un environment para que la podamos utilizar. Si nosotros llegaramos a mencionar una variable y no está definida da error.
Sabiendo esto, necesitamos dos procedimientos: definir una variable con un valor, y una función que dada una variable me de su valor.
Extendamos nuestro tipo de Expr

```
type Id = String
data Expr = EConstNum Int
          | EConstBool Bool
          | EAdd Expr Expr
          | ELet Id Expr Expr
          | EVar Id
```

Nota: Id es básicamente la forma que vamos a identificar las variables, en este caso con un String. Ej. "x"
¿Qué es lo que tenemos que refactorizar? Tenemos que aceptar los casos de EVar y ELet en nuestro evalExpr por lo tanto

```
evalExpr :: Expr -> Val
evalExpr (EConstNum n)  = VN n
evalExpr (EConstBool b) = VB b
```

```
    evalExpr (ELet x e1 e2) = ?
    evalExpr (EVar x) = ?
    evalExpr (EAdd n m) = addVal (evalExpr n) (evalExpr m)
```

¿Cúal es el problema de esto? ¡No tenemos en el environment donde queremos almacenar la definición de la variable x!
Importante: Recordar que el Environment de Haskell inicialmente vamos a enviarlo vacío. Si llegase a haber una acción sobre él, devolvemos una instancia nueva.
Ej.: Si tenía x = 2 y ahora quiero agregar z = 4, voy a tener lo anterior + z = 4 en un nuevo ambiente. Sería una especie de asegura de especificación.
Por lo tanto, definamos **de qué manera** vamos a usar el Environment.

```

    data Env a = EE [(Id, a)]

    emptyEnv :: Env a
    emptyEnv = EE []

    lookupEnv :: Env a -> Id -> a
    lookupEnv (EE e) x =
    case lookup x e of
        Just y  -> y
        Nothing -> error ("La variable " ++ x ++ " no está definida.")

    extendEnv :: Env a -> Id -> a -> Env a
    extendEnv (EE e) x a = EE ((x, a) : e)

    data Val = VN Int | VB Bool | VS String deriving Eq -> Agregamos VS String (para el ID)

    instance Show Val where
    show = showVal
        where
            showVal :: Val -> String
            showVal (VN n) = show n
            showVal (VB b) = show b
            showVal (VS s) = show s -> Agregamos
```

Ahora sí, extendamos nuestro evalExpr

```
    evalExpr :: Expr -> Env Val -> Val
    evalExpr (EConstNum n) _  = VN n
    evalExpr (EConstBool b) _ = VB b
    evalExpr (EAdd n m) env = addVal (evalExpr n env) (evalExpr m env)
    evalExpr (EVar x) env = lookupEnv env x
    evalExpr (ELet x e1 e2) env = ?
```

¿Cómo definimos x con la expresión 1 y la expresión 2?
Recordemos nuevamente, que acá tenemos que ir pisando el environment, que x es una expresión y en el environment tenemos que guardar su **valor** por lo tanto no hay que olvidarnos de obtener el valor de x.

```
    evalExpr (ELet x e1 e2) env =
                let v = evalExpr e1 env
                    env' = extendEnv env x v
----------------in evalExpr e2 env'
```

Luego, agregamos el show a la aplicación del let y el var

```
    ...lo anterior
    showExpr (ELet x e1 e2) = "let " ++ x ++ " =" ++ show e1 ++ " in " ++ show e2
    showExpr (EVar x) = x
```

Ahora, ejecutemos alguna instrucción que haya funcionado antes
$evalExpr\ (EAdd\ (EConstNum\ 1)\ (EConstNum\ 2))\ emptyEnv \rightarrow 3$
Probemos ahora sí ejecutar expresiones del tipo let y var. Definamos una variable x = 3, hagamos y = 4 y luego x + y

- Sintaxis
  - Input: (ELet "x" (EConstNum 3) (ELet "y" (EConstNum 4) (EAdd (EVar "x") (EVar "y"))))
  - Output Esperado: let x = 3 in let y = 4 in (x+y)

- Código
  - Input: evalExpr (ELet "x" (EConstNum 3) (ELet "y" (EConstNum 4) (EAdd (EVar "x") (EVar "y")))) emptyEnv
  - Output: 7

Simulando un Lenguaje Imperativo con nuestro Intérprete

¿Qué es lo que tiene un lenguaje imperativo que no tiene un lenguaje funcional? la asignación.
La asignación en un lenguaje imperativo se hace comúmente de la siguiente forma

- El nombre de la variable se busca su address en memoria.
- Se hace una escritura sobre ese address con un nuevo valor manteniendo el mismo nombre.

Entonces de alguna manera necesitamos simular una memoria. Para eso definamos el tipo y las funciones que necesitamos (los dió la catedra).

```
module Memory(Addr, Mem, emptyMem, freeAddress, load, store) where

type Addr = Int
data Mem a = MM [(Addr, a)]

instance Show (Mem a) where
show (MM _) = "<memoria>"

emptyMem :: Mem a
emptyMem = MM []

freeAddress :: Mem a -> Addr
freeAddress (MM []) = 0
freeAddress (MM xs) = maximum (map fst xs) + 1

load :: Mem a -> Addr -> a
load (MM xs) a =
case lookup a xs of
    Just b  -> b
    Nothing -> error "La dirección de memoria no está."

store :: Mem a -> Addr -> a -> Mem a
store (MM xs) a b = MM ((a, b) : xs)
```

Luego, modificamos las acciones que pueden tener nuestras expresiones en un lenguaje imperativo: asignar un valor nuevo a una variable y ejecutar una secuencia de expresiones

```
type Id = String
data Expr = EConstNum Int
          | EConstBool Bool
          | EAdd Expr Expr
          | ELet Id Expr Expr
          | EVar Id
          | EAssignID Expr
          | ESeq Expr Expr
```

Entonces ahora sí ¿qué cambia de evalExpr?

- Ya no recibimos el valor por una Env sino que ahora recibimos un address.
- El valor de un address particular está atado a la memoria y depende de esa memoria.

```
evalExpr :: Expr -> Env Addr -> Mem Val -> (Val, Mem Val)
evalExpr (EVar x) env mem = (load mem(lookupEnv x), mem)
evalExpr (EAdd e1 e2) env mem =
        let (v1, mem') = evalExpr e1 env mem
------------let (v2, mem'') = evalExpr e2 env mem'
------------in (addVal v1 v2, mem'')
----evalExpr (ELet x e1 e2) env mem =
------------let addr = freeAddr mem
```

```
            (val, mem') = evalExpr e1 env mem
                mem'' = store mem' addr v
            in evalExpr e2 env' mem''
    evalExpr (EAssign x expr) env mem =
            let addr = lookupEnv env x
            (v, mem') = evalExpr e env mem
            mem'' = store mem' addr v
            in (v, mem'')

    evalExpr (ESeq e1 e2) env mem =
        let (v1, mem') = evalExpr e1 env mem
        in eval e2 env mem'
```

Agregando los condicionales a nuestro intérprete

```
    evalExpr (EIf c t e) env mem =
        case eval c env mem of
            (VB b, m) -> evalExpr (if b then t else e) env m
            (VN n, m) -> error "Cond no bool"
```

Nota: Es posible encontrar este código en la carpeta de teóricas hecho por mí.

Características Funcionales

Volvamos a nuestro lenguaje púramente funcional sin memoria ni direcciones.

```
    evalExpr :: Expr -> Env Val -> Val
    evalExpr (EConsNum n) _ = h
    evalExpr (EConsBool b) _ = b
    evalExpr (EAdd e1 e2) _ = evalExpr e1 env 'addVal' evalExpr e2 env
    evalExpr (EVar x) env = lookup env x
    evalExpr (ELet x e1 e2) env = evalExpr e2 (extendEnv env x (evalExpr x env))
    evalExpr (EIf c t e) env = evalExpr (if isTrue(evalExpr c env) then t else e) env
        where isTrue (v b x) = x
              isTrue _ = False
```

Prácticamente todos los lengaujes funcionales están basados en el cálculo-λ
El cálculo-λ es un lenguaje que tiene solamente tres constructores.

- EVar Id $--x$
- ELam Id Expr $--\backslash x \to e$
- EApp Expr Expr $--e1\ e2$

Ahora queremos evaluar cosas como $(\backslash x \to x + x)$
Por lo tanto en nuestro data Val agreguemos el constructor VFunction, y en las expresiones que queremos evaluar agreguemos la composición y las funciones lambda.

```
    data Val = VN Int | VB Bool | VFunction Id Expr

    data Expr = ...
              | ELan Id expr — \x -> c
              | EApp expr expr — e1(e2)
```

Ahora agreguemos el cuerpo de esas operaciones de expresiones

```
    ...
    evalExpr (ELam x e) env = VFunction x e
    evalExpr (EApp e1 e2) env =
        let v1 = evalExpr e1 env
        let v2 = evalExpr e2 env
        in case v1 of
            VFunction x e -> evalExpr e (extendEnv env x v2)
                          _ -> error "aplicando una no funcion"
```

Importante: la e es el cuerpo de la función mientras que (extendEnv env x v2) es el pasaje de parámetros a una función lambda. Los parámetros se pasan a través del environment.
Con esta implementación tenemos un problema, si queremos hacer lo siguiente

```
let suma = \x -> \ y -> x + y in
let f = suma 5 in
let x = 0 in
    f 3
```

Tenemos el problema que nosotros querríamos que eso sea 8, pues estamos justamente aplicando parcialmente suma enviando un 5 pero esto da 3. ¿Por qué?
En Haskell, este comportamiento no sucede (porque ya lo contempla) pero es importante entender **qué es lo que pasa**.
Cuando hacemos let f = suma 5, f es un nombre a la función parcialmente aplicada suma que como primer argumento se le envía 5.
Sin embargo, cuando hacemos x = 0 f 3 lo que estamos haciendo es pisar el comportamiento de f diciendo ahora que la x que antes habíamos indicado vía (suma 5) ya no es suma 5, sino que ahora es suma 0.
Entonces, el paso a paso sería algo así

```
let suma = \x -> \y -> x + y in
let f = suma 5 in

f = \y -> x + y -> f no recuerda nada de lo definido parcialmente

let x = 0 in
    f 3

f = \3 -> x + 3 -> x tiene el valor de 0 en el entorno
f = 0 + 3 = 3
```

Importante: Las funciones parcialmente aplicadas deben recordar la definición en el ambiente que se les dió y qué parametros se les dieron a la hora de definirlas. De lo contrario, podría cambiar su comportamiento. Esto es conocido como **Closures**

Closures

Son funciones que recuerdan su ambiente a la hora de ser definidas. Cuando decimos ambiente decimos también con qué argumentos se definieron. Se utiliza para **capturar el entorno** en el que la función fue creada, permitiendo que acceda a variables externas en su ámbito local.
En lenguajes como JavaScript se ven de la siguiente forma

```
function crearContador() {
let contador = 0;
return function() {
    contador++;
    return contador;
}
}

const contador1 = crearContador();
console.log(contador1()); // 1
console.log(contador1()); // 2
```

Importante: Un closure mantiene la referencia al entorno donde fue creada, permitiendo que acceda a las variables de ese entorno en ese momento incluso después de que el entorno ya no exista. En nuestro intérprete debemos reemplazar las VFunctions por VClosures.
Los Closures suelen estar ligados a estrategias de evaluación, es decir, a las técnicas para evaluar aplicaciones de dos funciones.

Thunk

Un Thunk es una función que **no se evalúa hasta que realmente se lo necesite**. Es una técnica para diferir la evaluación de una expresión hasta que su valor sea necesario.
Ej.: let exp = 2 + 3 no devuelve 5 hasta que realmente se lo necesite.
En lenguajes como JavaScript se ven de la siguiente forma: let thunk = () $\implies$ 2 + 3; entonces cuando la necesitamos hacemos básicamente thunk()
Importante: Un Thunk difiere la evaluación de una expresión pero no capturan variables de su entorno.

Estrategias de Evaluación de dos funciones

- Llamada por valor (call-by-value)
  - Se evalúa e1 hasta que sea un closure.
  - Se evalúa e2 hasta que sea un valor.
  - Se evalúa el cuerpo de la función utilizando el resultado de e2 (valor)
  - El parámetro queda ligado al valor de e2.
- Llamada por nombre (call-by-name)
  - Se evalúa e1 hasta que sea un closure.
  - Se evalúa el cuerpo de la función.
  - El parámetro queda ligado a la expresión e2 sin evaluar (en este contexto, es un thunk)
  - **Cada vez** que se usa el parámetro se evalúa la expresión e2.
- Llamada por necesidad (call-by-need)
  - Se evalúa e1 hasta que sea un closure..
  - Se evalúa el cuerpo de la función.
  - El parámetro queda ligado a la expresión e2 sin evaluar.
  - **La primera vez** que el parámetro se necesita, se evalúa e2.
  - Se guarda el resultado para evitar evaluar e2 nuevamente (necesito memoria)

Importante: En colores, están los pasos que comparten entre las diferentes estrategias.

Aplicando Closures a nuestro Intérprete (Call By Value)

Cambiemos entonces, VFunction por VClosures.

```
data Val = VN Int | VB Bool | VClosure Id Expr (Env Val)
```

¡Importante el env! Es la idea de los closures.
Ahora modifiquemos el evalExpr para utilizar closures.

```
evalExpr :: Expr -> Env Val -> Val
evalExpr (ELam x e) env = VClosure x e env
evalExpr (EApp e1 e2) env =
    let v1 = eval e1 env
    let v2 = eval e2 env
        in case v1 of
            VClosure x e env' -> evalExpr e (extendEnv env' x v2)
                        _ -> error "Aplicando una no función"
```

¿Por qué esta estrategia es Call By Value? Porque al evaluar al expresión e estamos directamente pasando el valor de evaluar e2 de antemano.

Aplicando Closures a nuestro Intérprete (Call By Name)

Como acá necesitamos sí o sí diferir de no calcular el valor de e2 de antemano, entonces acá **evaluamos expresiones no valores**.

```
data Thunk = TT Expr (Env Thunk)
data Val = ...
         | VClosure Id Expr (Env Thunk)

evalExpr :: Expr -> Env Thunk -> Val
evalExpr (EVar x) env =
                case lookup env x f
                tt e env' -> eval e env'
evalExpr (EApp e1 e2) env =
    let v1 = eval e1 env
        in case v1 of
            VClosure x e env' -> evalExpr e (extendEnv env' x) (tt e2 env)
                        _ -> error "Aplicando una no función"
```

¿Por qué esta estrategia es Call By Name? Porque en ningún momento evalúamos e2 si no la necesitamos. Sino que solamente se evalúa obteniendo su valor cuando se necesita en su ambiente dado.

Aplicando Closures a nuestro Intérprete (Call By Need)

En call-by-need hay dos tipos de valores

- Valores que ya están calculados (atómicos): enteros, booleanos, closures.
- Valores pendientes de ser evalúados (thunks).

Para utilizar call by need tenemos que tener en cuenta

- Asociación de Identificadores a Direcciones: Esto quiere decir que el entorno (Env) asocia los nombres de las variables (identificadores) con **direcciones de memoria**.
- Asociación de Direcciones a Valores: Esas direcciones de memoria de los identificadores tienen valores finales o thunks.
- Evaluar una expresión: Nos da como resultado su valor final.

Ej conceptual: let x = (2 + 3) * 4
En este ejemplo x es básicamente un nombre o identificador que almacena un thunk dentro (2+3) * 4.
Si en algun momento se llegase a utilizar, x toma el valor de 20 y lo guarda en la memoria.

Sistemas Deductivos

Es una herramienta matemática que formaliza el lenguaje de la demostración.
Existen muchos sistemas deductivos, acá vamos a ver el de deducción natural que es uno parecido a la forma de pensar del humano.
Está dado por un conjunto de **axiomas** y **reglas de inferencia** que tienen la siguiente estructura

$$\frac{}{\langle \text{ axioma } \rangle}\ \langle \text{ nombre } \rangle \qquad \frac{\langle premisa_0 \rangle\ \langle premisa_1 \rangle ...\ \langle premisa_n \rangle}{\langle conclusion \rangle}\ \langle \text{ nombre de la regla } \rangle$$

Axioma: Afirmaciones básicas que se asumen como verdaderas. No es posible deducirlas de otras afirmaciones.
Reglas de Inferencia: Permiten derivar afirmaciones (teoremas) a partir de axiomas y otras afirmaciones.

Árbol de Derivación

- Las premisas son hojas.
- Los nodos representan afirmaciones.
- La raíz es la afirmación que se quiere probar.
- Las ramas representan las reglas de inferencias que conectan a las afirmaciones.

Si llegamos a una prueba correcta, **las hojas son axiomas**.

Afirmación Derivable (Teorema)

Una afirmación es derivable si existe alguna derivación sin premisas que la tiene como conclusión.

Fórmulas

- Cualquier variable proposicional es una fórmula.
- Si ρ es una fórmula, entonces $\neg\rho$ es una fórmula.
- Si ρ y σ son fórmulas, $\rho \wedge \sigma$ es una fórmula.
- Si ρ y σ son fórmulas, $\rho \vee \sigma$ es una fórmula.
- Si ρ y σ son fórmulas, $\rho \implies \sigma$ es una fórmula.
- Si ρ y σ son fórmulas, $\rho \iff \sigma$ es una fórmula.

Al ser un conjunto inductivo, viene provisto de

- Esquema de prueba para probar propiedades sobre ellos **inducción estructural**.
- Esquema de recursión para definir funciones sobre el conjunto **recursión estructural**.

Gramática de la Lógica Proposicional

Las fórmulas son las expresiones que se pueden generar a partir de la siguiente gramática

$$\tau, \sigma, \rho... ::= P \mid \bot \mid (\tau \wedge \sigma) \mid (\tau \implies \sigma) \mid (\tau \vee \sigma) \mid \neg\tau$$

Llamamos **contexto** a un conjunto finito de fórmulas.

Valuación

Una valuación es una función $v : \mathcal{V} \implies \{V, F\}$ que asigna valores de verdad a las variables proposicionales.
Una valuación satisface una proposición τ si $v \vDash \tau$ donde:

- $v \vDash P$ *sii* $v(P) = V$
- $v \vDash \neg\tau$ *sii* $v \nvDash \tau$
- $v \vDash \tau \vee \sigma$ *sii* $v \vDash \tau$ *o* $v \vDash \sigma$
- $v \vDash \tau \wedge \sigma$ *sii* $v \vDash \tau$ *y* $v \vDash \sigma$
- $v \vDash \tau \implies \sigma$ *sii* $v \nvDash \tau$ *o* $v \vDash \sigma$
- $v \vDash \tau \iff \sigma$ *sii* $v \vDash \tau$ *sii* $v \vDash \sigma$

Nota: Una valuación es una fila de la tabla de verdad.
Nota 2: $\vDash$ se lee como **satisface**
A modo de ejercicio escribamos uno de estas valuaciones con árboles de derivación a nivel de sistema deductivo:

$$\frac{v \vDash \tau \; v \vDash \sigma}{\langle \; v \vDash \tau \wedge \sigma \; \rangle} \quad \langle \wedge \rangle$$

Equivalencia Lógica y Tipos de Fórmulas

Dadas dos fórmulas τ y σ:

- τ es lógicamente equivalente a σ cuando $v \vDash \tau$ sii $v \vDash \sigma$ para toda valuación v.

Una fórmula τ es:

- Una tautología si $v \vDash \tau$ para toda valuación v.
- Satisfactible si existe una valuación v tal que $v \vDash \tau$
- Insatisfactible si no es satisfactible.

Un conjunto de fórmulas Γ es

- Satisfactible si existe una valuación v tal que $\forall\tau \in \Gamma$ se tiene $v \vDash \tau$
- Insatisfactible si no es satisfactible.

Teorema de la Insatisfactibilidad

Una fórmula τ es una tautología sii $\neg\tau$ es insatisfactible.

Sistema Deductivo basado en Reglas de Prueba

- Secuente (expresión que incluye conclusión): $\tau_1, \tau_2, ..., \tau_n \vDash \sigma$
  - Denota que a partir de asumir que el conjunto de fórmulas $\tau_1, \tau_2, ..., \tau_n$ son tautologías, podemos obtener una prueba de la validez de σ
- Reglas de prueba:

$$\frac{\Gamma_1 \vDash \tau_1 \ldots \Gamma_n \vDash \tau_n}{\Gamma \vDash \sigma : \mathit{conclusion}} \text{ nombre de la regla}$$

Convenciones de Notación

- Omitimos los paréntesis más externos de las fórmulas: $\tau \wedge \neg(\sigma \vee \rho) = (\tau \wedge \neg(\sigma \vee \rho))$
- La implicación es asociativa a la derecha: $\tau \implies \sigma \implies \rho = (\tau \implies (\sigma \implies \rho))$
- Los conectivos $\wedge, \vee$ **no** son conmutativos ni asociativos
  - $\tau \vee (\sigma \vee \rho) \neq (\tau \vee \sigma) \vee \rho$
  - $\tau \wedge \sigma \neq \sigma \wedge \tau$

Consecuencia Lógica

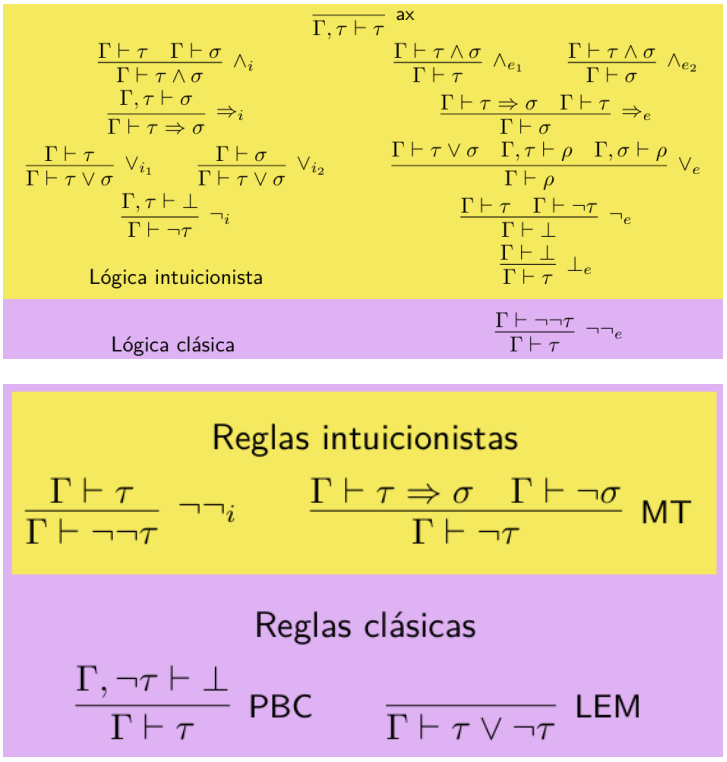
- $\bot$ es siempre falso
- $v \vDash \emptyset$
- $\emptyset \vDash T$ todas las tautologías

Lógica Intusionista (NJ) vs Lógica Clásica (NK)

Las reglas de la lógica intusionista están contenidas en las reglas de la lógica clásica.
Algunas reglas que podemos aplicar en la **lógica clásica** que no podemos usar en la Lógica Intusionista

- PBC (proof by contradiction): No es derivable, es decir, no se puede deducir.
- LEM (law of excluded middle): Es un axioma, no tiene premisas y por lo tanto no se puede probar.
- $\neg\neg e$ (doble negation elimination)

Las reglas **PBC**, **LEM** y $\neg\neg e$ son equivalentes.



Weakening (Debilitamiento) y Strengthening (Fortalecimiento)

Weakening: Lo voy a decir en criollo, su demostración se hace por inducción, pero básicamente es que si algo se cumple solo con τ entonces si agrego otras hipótesis también seguirá valiendo.
Se dice que es Weakening porque estamos haciendo más débil la prueba, esto es porque a mayor cantidad de información que tiene el contexto es más dificil de probar.

$$\frac{}{\Gamma \vDash \sigma} \implies \frac{}{\Gamma, \tau \vDash \sigma}$$

Strengthening: Es lo contrario a Weakening, la idea es que podemos ir sacando hipótesis que no nos sirven y poder seguir probando que vale. Se utiliza mucho en algo llamado Cálculo Lambda que veremos luego.
Se dice que es Strengthening porque estamos haciendo más fuerte la prueba, esto es porque a menor cantidad de información que tiene el contexto es más fácil de probar.

Correctitud (Soundness) y Completitud

Importante: Utilizamos $\vDash$ para hablar de satisfacibilidad en el contexto de **semántica** mientras que utilizamos $\vdash$ para indicar que una fórmula se puede derivar en el contexto de **sintáxis**.

Correctitud y Completitud responde a la siguiente pregunta: ¿cuál es la relación entre la sintáxis y la semántica?

- **Sintáxis**: Conjunto de fórmulas τ tal que $\neg\tau$ es un secuente válido.
- **Semántica**: Conjutno de fórmulas τ tal que $v \vDash \tau$ para toda valuación v. Ej: Tautologías.

$$\Gamma \vDash T \iff \Gamma \vdash T$$

Corrección
$\vDash \tau$ secuente válido implica que τ es tautología.

- $\sigma_1, \sigma_2, \ldots, \sigma_n \vdash \tau$ secuente válido implica que $\sigma_1, \sigma_2, \ldots, \sigma_n \vDash \tau$

Para demostrar: Hacemos inducción en la estructura de la prueba analizando por casos la última regla aplicada en la prueba.
Caso base, la última regla fue un axioma.
Casos recursivos, el resto de reglas $\wedge_i, \wedge_{e1}, etc$
Completitud τ tautología implica que $\vdash \tau$ es secuente válido.

- $\sigma_1, \sigma_2, \ldots, \sigma_n \vDash \tau$ implica que $\sigma_1, \sigma_2, \ldots, \sigma_n \vdash \tau$ es un secuente válido.

Para demostrar: Usamos el contrarrecíproco: si $P \implies Q$ vale que $\neg Q \implies \neg P$
Luego probar estos dos lemas

- Si $\Gamma \nvdash \tau$ entonces $\Gamma \cup \{\neg\tau\}$ es consistente (sale por contrarrecíproco).
- Si Γ es consistente, entonces tiene modelo. Ej.: Γ es satisfactible (es más dificil).

Consecuencia Semántica

Sean $\tau_1, \tau_2, \ldots, \tau_n, \sigma$ fórmulas de la lógica proposicional.
$\tau_1, \tau_2, \ldots, \tau_n \vDash \sigma \iff \forall\ v\ valuacion\ ((v \vDash \tau_1 \wedge v \vDash \tau_2 \wedge \cdots \wedge v \vDash \tau_n) \implies (v \vDash \sigma))$
En crillo: Vale que $\tau_1, \tau_2, \ldots, \tau_n \vDash \sigma \iff$ para toda valuación v tal que v satisface a todas las hipótesis $v \vDash \tau_i$ para toda $i \in 1, \ldots, n \implies v \vDash \sigma$

Consistencia

Decimos que Γ es consistente si $\Gamma \nvdash \bot$, es decir, Γ es consistente si no se puede derivar una contradicción a partir de él.

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.: $devolverFalsoSiVerdadero\ :\ Bool \rightarrow Prelude.Maybe\ Bool$

El Maybe deja la puerta abierta a un valor posible "Nothing". Entonces tenemos dos casos: Si me envian 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.: $devolverRepresentacionIntBool\ ::\ Either\ Int\ Bool\ \rightarrow\ Int$

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

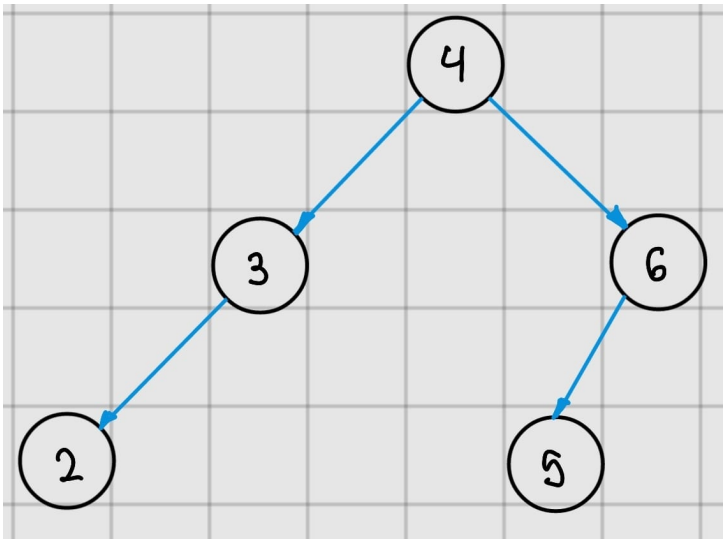
Se utiliza $data\ nombretipo\ tipo\ =\ Tipo\ 1\ |\ Tipo\ 2$ El | se interpreta como **o bien**

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: $suma\ ::\ (Int, Int) -> Int$

Por la definición de curry necesitamos que por cada argumento, haya una función que lo devuelva, por lo tanto el resultado sería algo así $suma :: Int- > Int- > Int$.
Veamos el tipo de función que queremos currificar: $((a, b)- > c)$, esto lo queremos llevar a $a- > b- > c$.
Por lo tanto nuestra función curry sería algo así:

```
curryOwn :: ((a, b) -> c) -> a -> b -> c
curryOwn f a b = f (a, b)
```

Entonces, digamos que queremos hacer la suma currificada.

```
sumTuple :: (Float, Float) -> Float
sumTuple (x, y) = x + y

sumarCurry :: Float -> Float -> Float
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 -¿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 $a- > b- > c$ debo llevarlo a $(a, b)- > c$

```
uncurryOwn :: a -> b -> c -> ((a, b) -> c)
uncurryOwn f (a, b) = f a b

sumarUncurry :: (a, b) -> c
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 algun criterio, es decir, podemos aplicar $> < =$ etc.
- Eq a: Indica que el parámetro a se puede igualar, es decir, podemos aplicar $=$

Foldr

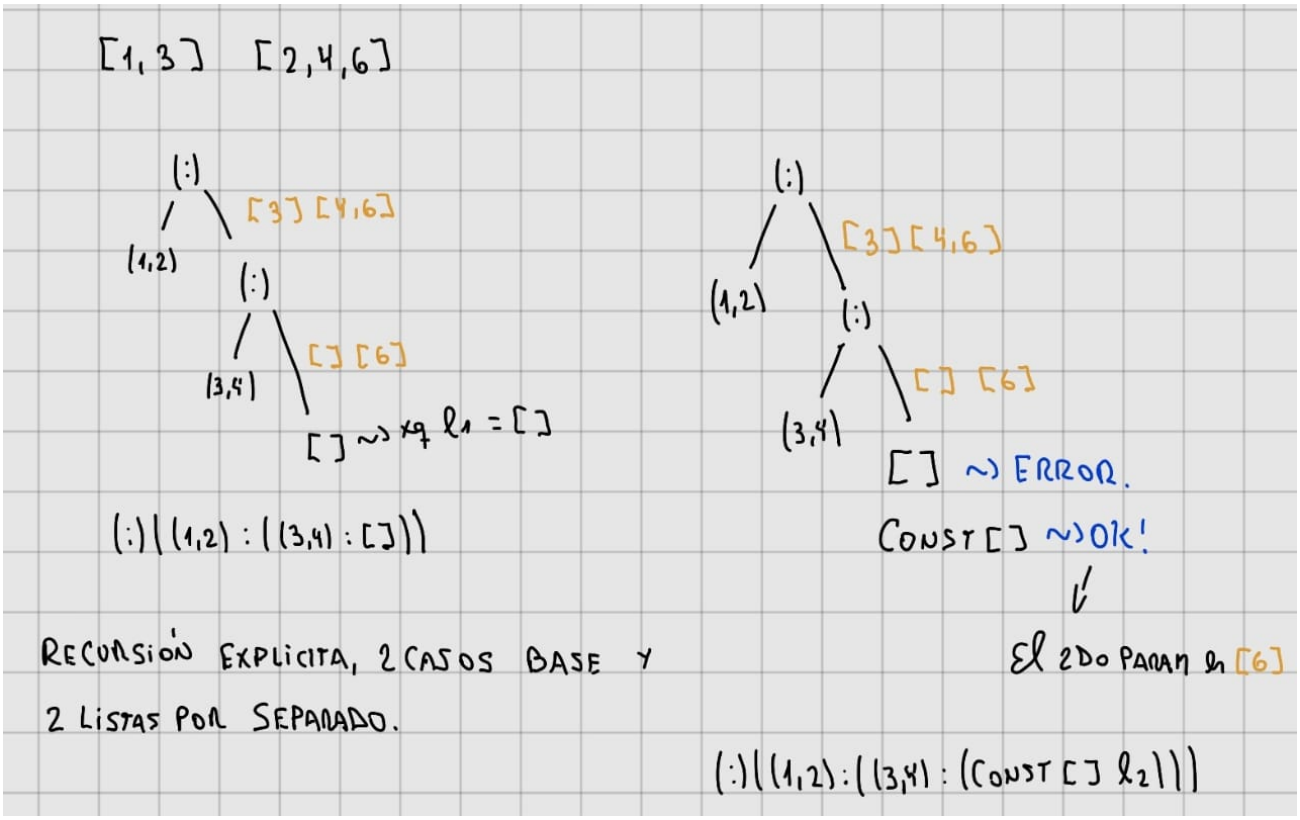
```
// Solo recorre listas de tipo a. Es decir, devuelve la suma de los elementos.
sumFoldrlist :: Num a => [a] -> a
sumFoldrlist = foldr (\x ac -> x + ac) 0

//Recorre tipos plegables. Acá no nos limitamos solo a listas, porque véase que usamos t a en vez de [a]
sumFoldr :: (Foldable t, Num a) => t a -> a
sumFoldr = foldr (\x ac -> x + ac) 0
```

Foldr, el árbol de recursión y más de una lista

Uno de los problemas más normales es tener que enviar más de una lista a procesar a una función dada en Haskell y realizar recursión estructural.
Veamos el siguiente ejercicio: Arme pares de la forma [(a, b)] usando recursión estructural.
Esto es súper simple si lo hacemos sin recursión estructural porque nos queda algo así

```
armarPares :: [a] -> [b] -> [(a, b)]
armarPares [] _ = []
armarPares _ [] = []
armarPares (x:xs) (y:ys) = (x, y) : armarPares xs ys
```

¿Cómo hacemos esto con foldr? Veamos el árbol recursivo.



El error en la recursión estructural vendría del lado de que como estamos recorriendo dos listas a la vez, y mi llamado recursivo es del tipo $[b] \to [(a, b)]$ no puedo devolver [] entonces lo que hago es devolver *const* [] y se aplica parcialmente al argumento que sería la segunda lista *const* [] $l2$ y como la primera lista está vacía entonces devuelve [].
Esto es súper importante a tener en cuenta, porque si mandamos más de un argumento en la recursión, recordar el concepto de curry.

Flip

Toma dos parámetros y devuelve una función que los devuelve en el orden inverso.
Es decir: $(a \to b \to c) \to b \to a \to c$
Luego: $flip\ f\ a\ b\ =\ f\ b\ a$

Función identidad

Devuelve el mismo valor aplicado a la función.
Es decir: $(a \to b) \to a \to b$
Luego: $\$f\ a\ =\ f\ a$

Función constante

Devuelve un valor enviado sin aplicarle ninguna función.
Es decir: $a \to b \to a$
Luego: $const\ a\ b\ =\ a$

Reduciendo expresiones elegantemente

- $filter(\backslash x \to length\ x > 3)$
  - ¿Puede hacerse algo mejor? No. Porque a x si o sí necesitamos aplicarle una función.
- $filter(\backslash x \to x > n)$
  - ¿Puede hacerse algo mejor? Sí. $filter(> n)$
- $filter(\backslash x \to mod x 2 / = 0)$
  - ¿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 $map\ (map\ toUpper)$
- $doblarElementos.filtrarPares$
  - ¿Puede hacerse algo mejor? No. Esto es el equivalente a un lenguaje imperativo hacer doblarElementos(filtrarPares(lista))

¿Qué hacen las siguientes funciones compuestas?

```
flip ($) 0 id

(==0) . (flip mod 2)
```

Primero veamos que hace flip mod 2.
mod 2 es notación infija (Integral a ⇒ 2 -> a -> a), entonces lo que está diciendo es que si le paso cualquier número va a hacer mod 2 x, y nosotros por lo que yo entiendo es que queremos ver si es par.
Por lo tanto, lo primero que haríamos es invertir los argumentos de mod 2 con flip (Integral a ⇒ a -> 2 -> a), entonces quedaría algo como mod x 2 donde el x lo tenemos que enviar nosotros.
Luego, se compone la función de mod x 2 == 0 esperando solo un argumento donde verifica si efectivamente un número dado es par.
Entonces, (Integral a ⇒ x -> Bool)

```
map f = ((:) . f)
```

Lo que hace esta función es básicamente aplicar una función f a todos los elementos y agregarlos a una lista particular.

Dado ["hola", "abc"] quiero devolver ["cba", "aloh"]. Es decir, dar vuelta cada caracter de cada palabra y ademas dar vuelta las palabras.

```
reverseAnidado :: ["String"] -> ["String"]
reverseAnidado = reverse . (map . reverse)
```

Lo primero que hacemos es hacer un map haciendo reverse por cada caracter de la lista. Luego, reordenamos las palabras en sí.
El tipo de reverse es: [a] -> [a] pero con los elementos al revés. Entonces, por cada palabra (map) hacemos un reverse y las guardamos.
Finalmente, nos queda algo así ["aloh", "cba"], nos queda dar vuelta eso, entonces hacemos nuevamente un reverse de toda la lista. ["cba", "aloh"].

```
listacomp f xs p = [f x | x <- xs, p x]
listacomp f xs p = map f (filter p xs)
```

Ejercicios Foldl

1. Definir la función sumasParciales que dada una lista de números devuelve otra de la misma longitud que tiene en cada posicion 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 → a → 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 asi → [1, 2, 3]
  - 1:[] = [1]
  - 2 + (head [1]) : [1] = [3, 1]
  - 3 + (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:

```
sumasParciales :: Num a => [a] -> [a]
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 $a \to b \to c$ 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: $map\ (uncurry f)(lista)$

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

donde la función armarPares tiene la siguiente pinta

```
armarPares :: [a] -> [b] -> [(a, b)]
armarPares _ [] = []
armarPares [] _ = []
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 $Int \to Int \to Int$ y yo voy a mandar $(Int, Int) \to Int$.

```
sumaMat :: (Int -> Int -> Int) -> [[Int]] -> [[Int]] -> [[Int]]
sumaMat _ [] _ = []
sumaMat _ _ [] = []
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.

```
sumaMat :: (Int -> Int -> Int) -> [[Int]] -> [[Int]] -> [[Int]]
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.