

# Teórica 1

## Paradigmas de Programación

### Esquemas de recursión Tipos de datos inductivos

1er cuatrimestre de 2024

Departamento de Computación  
Facultad de Ciencias Exactas y Naturales  
Universidad de Buenos Aires

## Funciones anónimas

### Notación "lambda"

Expresión de la forma:  $x \rightarrow e$

Una función que recibe un parámetro  $x$  y devuelve  $e$ .

Abreviatura

$(\lambda x_1 \rightarrow (\lambda x_2 \rightarrow \dots (\lambda x_n \rightarrow e))) \equiv (\lambda x_1 x_2 \dots x_n \rightarrow e)$

### Ejemplo

```
>> map (\ x -> (x, x)) 1
~> (1, 1)
```

1

2

## Currificación

```
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
```

## Currificación

### Ejemplo

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

```
suma' :: (Int, Int) -> Int
suma' (x, y) = x + y
```

Veremos que se puede demostrar lo siguiente

```
suma = curry suma'
suma' = uncurry suma
```

3

4

# Función map

## Map

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

## Ejemplo

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

What about map (+) [1,2,3]?  
map (+) [1, 2, 3] :: Num a => [a -> a]

## Ejemplo

```
>> map ($ 10) map (+) [1, 2, 3]
~> [11, 12, 13]
```

# Filter

## filter

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

# Ejercicios

- ▶ negativos :: [Float] -> [Float] tal que  
negativos xs contiene los elementos negativos de xs

```
negativos [] = []
negativos (x:xs) | x < 0 = x : negativos xs
                  otherwise = negativos xs
```

- ▶ noVacias :: [[a]] -> [[a]] tal que la lista noVacias xs  
contiene las listas no vacías de xs

```
noVacias [] = []
noVacias (x:xs) | length x > 0 = x : noVacias xs
                  otherwise = noVacias xs
```

# Ejercicio

¿Qué tipo tiene la expresión map filter? Hagamos un ejemplo de uso.

```
map filter :: [a -> Bool] -> [[a] -> [a]]
-- Definir predicados
predicadoPar :: Int -> Bool
predicadoPar x = x `mod` 2 == 0
predicadoMayorQueTres :: Int -> Bool
predicadoMayorQueTres x = x > 3
-- Definir la función
funcion :: [a -> Bool] -> [[a] -> [a]]
funcion = map filter
-- Crear una lista de predicados
predicados :: [Int -> Bool]
predicados = [predicadoPar, predicadoMayorQueTres]
-- Aplicar `funcion` a la lista de predicados para obtener filtros
filtros :: [[Int] -> [Int]]
filtros = funcion predicados
-- Definir una lista de números
listaDeNumeros :: [Int]
listaDeNumeros = [1, 2, 3, 4, 5, 6]
-- Aplicar cada filtro a la lista de números
resultado :: [[Int]]
resultado = map (\filtro -> filtro listaDeNumeros) filtros
-- Resultado esperado: [[2, 4, 6], [4, 5, 6]]
```

Pensemos algunas funciones sobre listas

- ▶ sumaL : la suma de todos los valores de una lista de enteros
- ▶ concat : la concatenación de todos los elementos de una lista de listas
- ▶ reverso : el reverso de una lista

Sea  $g :: [a] \rightarrow b$  definida por dos ecuaciones:

$$\begin{aligned} g [] &= \langle \text{caso base} \rangle \\ g (x : xs) &= \langle \text{caso recursivo} \rangle \end{aligned}$$

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

1. El caso base devuelve un valor fijo  $z$ .
2. El caso recursivo es una función de  $x$  y  $(g \text{ } xs)$

$$\begin{aligned} g [] &= z \\ g (x : xs) &= f \text{ } x \text{ } (g \text{ } xs) \end{aligned}$$

El caso recursivo no usa  $xs$

Recursión estructural

Ejemplos de recursión estructural

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

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

-- Insertion sort
isort :: Ord a => [a] -> [a]
isort [] = []
isort (x : xs) = insertar x (isort xs)
```

```
insertar :: Ord a => a -> [a] -> [a]
insertar x [] = [x]
insertar x (y:ys) | x <= y = x : y : ys
                  | otherwise = y : insertar x ys
```

Recursión estructural

```
sacarMinimo :: Ord a => [a] -> [a]
sacarMinimo [] = []
sacarMinimo (x:xs) | x == m = xs
                  | otherwise = x : sacarMinimo xs
                  where m = minimo (x:xs)
```

Ejemplo: recursión que **no** es estructural

```
-- Selection sort
ssort :: Ord a => [a] -> [a]
ssort [] = []
ssort (x : xs) = minimo (x : xs)
                : ssort (sacarMinimo (x : xs))
```

```
minimo :: Ord a => [a] -> a
minimo [x] = x
minimo (x:xs) = min x (minimo xs)
```

# Plegado de listas a derecha

La función `foldr` abstrae el esquema de recursión estructural:

```
foldr
  foldr f z [] = z
  foldr f z (x : xs) = f x (foldr f z xs)
```

¿Cuál es su tipo?

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

Toda recursión estructural es una instancia de `foldr`.

**(a -> b -> b):** Esta es una función que toma un elemento del tipo a y un acumulador del tipo b, y devuelve un nuevo acumulador del tipo b.  
**b:** Este es el valor inicial del acumulador.  
**[a]:** Esta es la lista de elementos del tipo a que vamos a procesar.  
**b:** Este es el valor final del acumulador después de que se ha procesado toda la lista.

13

# Plegado de listas a derecha

## Ejemplo — reverse con foldr

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

Es recursiva estructural. ¿Cómo la escribiríamos usando `foldr`?

```
reverse = foldr (\ x rec -> rec ++ [x]) []
```

Otras formas equivalentes:

```
reverse = foldr (\ x rec -> flip (++) [x] rec) []
reverse = foldr (\ x -> flip (++) [x]) []
reverse = foldr (\ x -> flip (++) ((: [])) x) []
reverse = foldr (\ x -> (flip (++) . (: [])) x) []
reverse = foldr (flip (++) . (: [])) []
```

15

# Plegado de listas a derecha

## Ejemplo — suma con foldr

```
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
```

Análogamente:

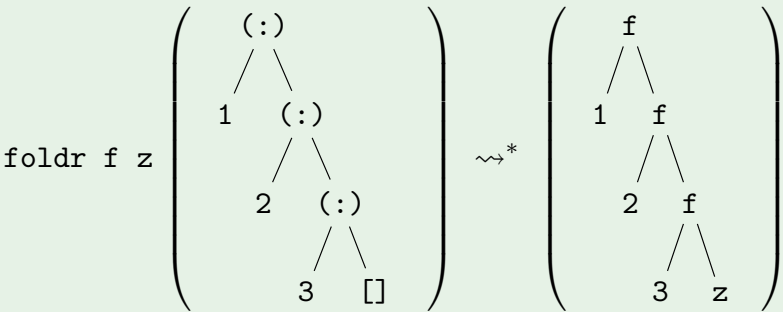
```
producto :: [Int] -> Int
producto = foldr (*) 1

and, or :: [Bool] -> Bool
and = foldr (&&) True
or  = foldr (||) False
```

14

# Plegado de listas a derecha

## Ilustración gráfica del plegado a derecha



En particular, se puede demostrar que:

```
foldr (:) [] = id
foldr ((:) . f) [] = map f
foldr (const (+ 1)) 0 = length
```

16

# Recursión primitiva

Sea  $g :: [a] \rightarrow b$  definida por dos ecuaciones:

$$\begin{aligned} g [] &= \langle \text{caso base} \rangle \\ g (x : xs) &= \langle \text{caso recursivo} \rangle \end{aligned}$$

Decimos que la definición de  $g$  está dada por **recursión primitiva** si:

- 1. El caso base devuelve un valor fijo  $z$ .
- 2. El caso recursivo se escribe usando (cero, una o muchas veces)  $x$ ,  $(g\ xs)$  y también  $xs$ , pero sin hacer otros llamados recursivos.

$$\begin{aligned} g [] &= z \\ g (x : xs) &= \dots x \dots xs \dots (g\ xs) \dots \end{aligned}$$

Similar a la recursión estructural, pero permite referirse a  $xs$ .

# Recursión primitiva

## Observación

- Todas las definiciones dadas por recursión estructural también están dadas por recursión primitiva.
- Hay definiciones dadas por recursión primitiva que no están dadas por recursión estructural.

## Ejemplo

Dado un texto, borrar todos los espacios iniciales.

```
trim :: String -> String
>> trim "  Hola PLP" == "Hola PLP"

trim [] = []
trim (x : xs) = if x == ' ' then trim xs else x : xs

Tratemos de escribirla con foldr.
```

17

trim' :: String -> String  
trim' = foldr (\x acc -> if null acc && x == ' ' then acc else x : acc) []

18

# Recursión primitiva

Escribamos una función `recr` para abstraer el esquema de recursión primitiva:

```
recr f z [] = z
recr f z (x : xs) = f x xs (recr f z xs)
```

¿Cuál es su tipo?

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

**Toda recursión primitiva es una instancia de `recr`.**

Escribamos `trim` ahora usando `recr`:

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

(a -> [a] -> b -> b): Es una función que toma un elemento de la lista, la cola de la lista y el resultado recursivo, y devuelve un nuevo resultado recursivo.  
b: Es el valor inicial  
[a]: Es la lista de elementos que vamos a procesar  
b: Es el valor final después de procesar la lista

# Recursión iterativa

Sea  $g :: b \rightarrow [a] \rightarrow b$  definida por dos ecuaciones:

$$\begin{aligned} g\ ac\ [] &= \langle \text{caso base} \rangle \\ g\ ac\ (x : xs) &= \langle \text{caso recursivo} \rangle \end{aligned}$$

## Recursión iterativa

Decimos que la definición de  $g$  está dada por *recursión iterativa* si:

- 1. El caso base devuelve el acumulador  $ac$ .
- 2. El caso recursivo invoca inmediatamente a  $(g\ ac'\ xs)$ , donde  $ac'$  es el acumulador actualizado en función de su valor anterior y el valor de  $x$ .

Ejemplos de recursión iterativa

```
-- Reverse con acumulador.
reverse' :: [a] -> [a] -> [a]
reverse' ac [] = ac
reverse' ac (x : xs) = reverse' (x : ac) xs

-- Pasaje de binario a decimal con acumulador.
-- Precondición: recibe una lista de 0s y 1s.
bin2dec' :: Int -> [Int] -> Int
bin2dec' ac [] = ac
bin2dec' ac (b : bs) = bin2dec (b + 2 * ac) bs

-- Insertion sort con acumulador.
isort' :: Ord a => [a] -> [a] -> [a]
isort' ac [] = ac
isort' ac (x : xs) = isort' (insertar x ac) xs
```

Escribamos una función foldl para abstraer el esquema de recursión iterativa:

```
foldl f ac [] = ac
foldl f ac (x : xs) = foldl f (f ac x) xs
```

¿Cuál es su tipo?

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

Toda recursión iterativa es una instancia de foldl.

(b -> a -> b): Esta es una función que toma un acumulador del tipo b y un elemento del tipo a, y devuelve un nuevo acumulador del tipo b.  
b: Este es el valor inicial del acumulador.  
[a]: Esta es la lista de elementos del tipo a que vamos a procesar.  
b: Este es el valor final del acumulador después de que se ha procesado toda la lista.

En general foldr y foldl tienen comportamientos diferentes:

```
foldr (★) z [a, b, c] = a ★ (b ★ (c ★ z))
foldl (★) z [a, b, c] = ((z ★ a) ★ b) ★ c
```

Si (★) es un operador asociativo y conmutativo, foldr y foldl definen la misma función. Por ejemplo:

```
suma      = foldr (+) 0      = foldl (+) 0
producto  = foldr (*) 1      = foldl (*) 1
and       = foldr (&&) True  = foldl (&&) True
or        = foldr (||) False = foldl (||) False
```

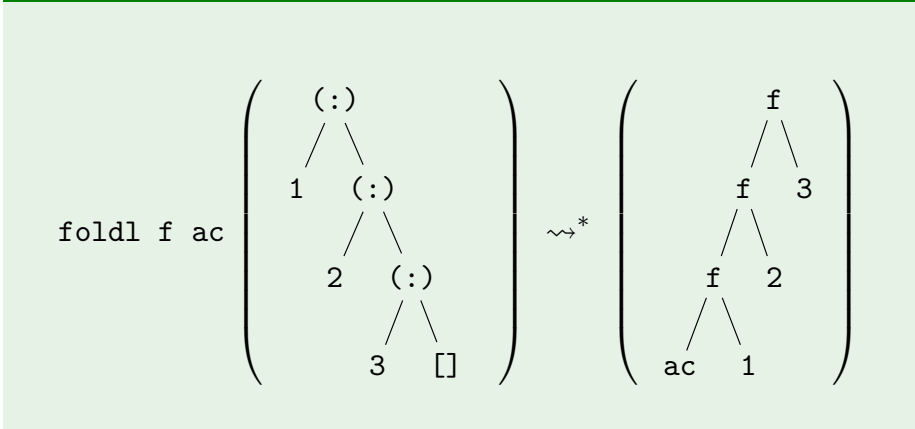
Ejemplo — pasaje de binario a decimal

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

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

# Plegado de listas a izquierda

## Ilustración gráfica del plegado a izquierda



En particular, se puede demostrar que:

```
foldl (flip (:)) [] = reverse
```

# Tipos de datos algebraicos

Conocemos algunos tipos de datos "primitivos":

```
Char  Int  Float  (a -> b)  (a, b)  [a]
String (sinónimo de [Char])
```

Se pueden definir nuevos tipos de datos con la cláusula `data`:

```
data Tipo = <declaración de los constructores>
```

# Para pensar

## Relación entre recursión estructural y primitiva

- 1. Definir `foldr` en términos de `recr`. (Fácil).
- 2. Definir `recr` en términos de `foldr`. (No tan fácil).  
Idea: devolver una tupla con una copia de la lista original.

## Relación entre recursión estructural e iterativa

- 1. Definir `foldl` en términos de `foldr`.
- 2. Definir `foldr` en términos de `foldl`.

Definir `foldr` en términos de `foldl` es generalmente más fácil y directo debido a la simplicidad de invertir una lista y aplicar `foldl`. Definir `foldl` en términos de `foldr` es más complicado y menos intuitivo porque requiere manipular la función de acumulación y la estructura de la lista de manera más elaborada.

# Tipos de datos algebraicos

## Ejemplo — tipos enumerados

Muchos constructores sin parámetros:

```
data Dia = Dom | Lun | Mar | Mie | Jue | Vie | Sab
```

Declara que existen constructores:

```
Dom :: Dia    Lun :: Dia    ...    Sab :: Dia
```

Declara además esos son los **únicos** constructores del tipo `Dia`.

```
esFinDeSemana :: Dia -> Bool
esFinDeSemana Sab = True
esFinDeSemana Dom = True
esFinDeSemana _   = False
```

```
>> esFinDeSemana Vie
~> False
```

## Tipos de datos algebraicos

### Ejemplo — tipos producto (tuplas/estructuras/registros/...)

Un solo constructor con muchos parámetros:

```
data Persona = LaPersona String String Int

Declara que el tipo Persona tiene un constructor (y sólo ese):

LaPersona :: String -> String -> Int -> Persona

nombre, apellido :: Persona -> String
fechaNacimiento  :: Persona -> Int
nombre          (LaPersona n _ _) = n
apellido        (LaPersona _ a _) = a
fechaNacimiento (LaPersona _ _ f) = f

rebecaGuber = LaPersona "Rebeca" "Guber" 1926
>> apellido rebecaGuber
↪ "Guber"
```

30

## Tipos de datos algebraicos

### Ejemplo

Un tipo puede tener muchos constructores con muchos parámetros:

```
data Forma = Rectangulo Float Float
           | Circulo Float

Declara que el tipo Forma tiene dos constructores (y sólo esos):

Rectangulo :: Float -> Float -> Forma
Circulo     :: Float -> Forma

area :: Forma -> Float
area (Rectangulo ancho alto) = ancho * alto
area (Circulo radio)         = radio * radio * pi
```

31

## Tipos de datos algebraicos

### Ejemplo

Algunos constructores pueden ser **recursivos**:

```
data Nat = Zero
        | Succ Nat

Declara que el tipo Nat tiene dos constructores (y sólo esos):

Zero  :: Nat
Succ  :: Nat -> Nat
```

¿Qué forma tienen los valores de tipo Nat?

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
...
```

32

## Tipos de datos algebraicos

Las funciones sobre tipos de datos con constructores recursivos normalmente se definen por recursión:

```
doble :: Nat -> Nat
doble Zero      = Zero
doble (Succ n) = Succ (doble n)
```

La siguiente ecuación, ¿define un valor de tipo Nat o es un error?

```
infinito :: Nat
infinito = Succ infinito
```

Respuesta:

- ▶ Depende de cómo se interpreten las definiciones recursivas.
- ▶ Generalmente nos van a interesar las estructuras finitas.
- ▶ En Haskell se permite trabajar con estructuras infinitas.   
Técnicamente hablando: en Haskell las definiciones recursivas se interpretan de manera *coinductiva* en lugar de *inductiva*.
- ▶ Ocasionalmente hablaremos de estructuras infinitas.

33



# Tipos de datos algebraicos

## Tipos de datos algebraico — caso general

En general un tipo de datos algebraico tiene la siguiente forma:

```
data T = CBase1 <parámetros>
      ...
      | CBasen <parámetros>
      | CRecursoivo1 <parámetros>
      ...
      | CRecursoivom <parámetros>
```

- ▶ Los constructores **base** no reciben parámetros de tipo T.
  - ▶ Los constructores **recursivos** reciben al menos un parámetro de tipo T.
  - ▶ Los valores de tipo T son los que se pueden construir aplicando constructores base y recursivos un número **finito** de veces y **sólo** esos.
- (Entendemos la definición de T de forma **inductiva**).

# Ejemplo: cuentas corrientes

```
type Cuenta = String
data Banco = Iniciar
           | Depositar Cuenta Int Banco
           | Extraer Cuenta Int Banco
           | Transferir Cuenta Cuenta Int Banco

bancoPLP = Transferir "A" "B" 3 (Depositar "A" 10 Iniciar)

saldo :: Cuenta -> Banco -> Int

saldo cuenta Iniciar = 0
saldo cuenta (Depositar cuenta' monto banco)
  | cuenta == cuenta' = saldo cuenta banco + monto
  | otherwise         = saldo cuenta banco
saldo cuenta (Extraer cuenta' monto banco)
  | cuenta == cuenta' = saldo cuenta banco - monto
  | otherwise         = saldo cuenta banco
saldo cuenta (Transferir origen destino monto banco)
  | cuenta == origen  = saldo cuenta banco - monto
  | cuenta == destino = saldo cuenta banco + monto
  | otherwise         = saldo cuenta banco
```

# Ejemplo: listas

Las listas son un caso particular de tipo algebraico:

```
data Lista a = Vacia | Cons a (Lista a)
```

O, con la notación ya conocida:

```
data [a] = [] | a : [a]
```

```
preorder Nil = []
preorder (Bin izq val der) = [val] ++ preorder izq ++ preorder der

postorder Nil = []
postorder (Bin izq val der) = postorder izq ++ postorder der ++ [val]

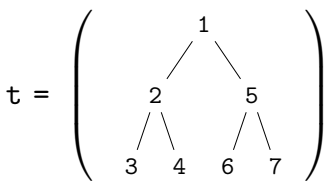
inorder Nil = []
inorder (Bin izq val der) = inorder izq ++ [val] ++ inorder der
```

# Ejemplo: árboles binarios

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

Definamos las siguientes funciones:

```
preorder :: AB a -> [a]
postorder :: AB a -> [a]
inorder :: AB a -> [a]
```



```
preorder t ~* [1, 2, 3, 4, 5, 6, 7]
postorder t ~* [3, 4, 2, 6, 7, 5, 1]
inorder t ~* [3, 2, 4, 1, 6, 5, 7]
```

# Ejemplo: árboles binarios

`insertar :: Ord a => a -> AB a -> AB a`

**Pre:** el árbol de entrada es un AB (sin repetidos).  
**Post:** el árbol resultante es un AB (sin repetidos) que contiene a los elementos del AB de entrada y al elemento dado.

```
insertar x Nil = Bin Nil x Nil
insertar x (Bin izq y der)
  | x < y      = Bin (insertar x izq) y der
  | x > y      = Bin izq y (insertar x der)
  | otherwise = Bin izq y der
```

# Recursión estructural

En el caso de las listas, dada una función `g :: [a] -> b`:

$$\begin{aligned} g [] &= \langle \text{caso base} \rangle \\ g (x : xs) &= \langle \text{caso recursivo} \rangle \end{aligned}$$

- decíamos que `g` estaba dada por recursión estructural si:
- ▶ El caso base devuelve un valor fijo `z`.
  - ▶ 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.

# Recursión estructural

La recursión estructural se generaliza a tipos algebraicos en general.  
Supongamos que `T` es un tipo algebraico.  
Dada una función `g :: T -> Y` definida por ecuaciones:

$$\begin{aligned} g (\text{CBase}_1 \langle \text{parámetros} \rangle) &= \langle \text{caso base}_1 \rangle \\ \dots & \\ g (\text{CBase}_n \langle \text{parámetros} \rangle) &= \langle \text{caso base}_n \rangle \\ g (\text{CRecursivo}_1 \langle \text{parámetros} \rangle) &= \langle \text{caso recursivo}_1 \rangle \\ \dots & \\ g (\text{CRecursivo}_m \langle \text{parámetros} \rangle) &= \langle \text{caso recursivo}_m \rangle \end{aligned}$$

- Decimos que `g` está dada por **recursión estructural** si:
1. Cada caso base se escribe combinando los parámetros.
  2. Cada caso recursivo se escribe combinando:
    - ▶ Los parámetros del constructor que no son de tipo `T`.
    - ▶ El llamado recursivo sobre cada parámetro de tipo `T`.
- Pero:
- ▶ Sin usar los parámetros del constructor que son de tipo `T`.
  - ▶ Sin hacer a otros llamados recursivos.

# Recursión estructural

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

## Ejemplo

Definamos una función `foldAB` que abstraiga el esquema de recursión estructural sobre árboles binarios.

```
foldAB :: b -> (b -> a -> b -> b) -> AB a -> b

foldAB cNil cBin Nil = cNil
foldAB cNil cBin (Bin i r d) =
  cBin (foldAB cNil cBin i) r (foldAB cNil cBin d)
```

`foldAB` toma tres parámetros:

- `cNil`, que es el valor que se devuelve cuando se encuentra un `Nil`
- `cBin`, que es una función que toma tres argumentos: el resultado del plegado del subárbol izquierdo (`i`), el valor del nodo actual (`r`), y el resultado del plegado del subárbol derecho (`d`)
- Un árbol `AB a` que se va a plegar

### Ejemplo

1. ¿Qué función es (foldAB Nil Bin)?
2. Definir mapAB :: (a -> b) -> AB a -> AB b usando foldAB.

1) (foldAB Nil Bin) es la función identidad del árbol binario, ya que devuelve el mismo árbol.

`foldAB Nil Bin :: AB a -> AB a`

2) `mapAB f = foldAB Nil Bin (\l r d -> Bin l (f r) d)`

42

### Casos degenerados de recursión estructural

Es recursión estructural (no usa la cabeza):

```
length :: [a] -> Int
length [] = 0
length (_ : xs) = 1 + length xs
```

Es recursión estructural (no usa el llamado recursivo sobre la cola):

```
head :: [a] -> a
head [] = error "No tiene cabeza."
head (x : _) = x
```

44