

# Programación Funcional (parte 2)

Paradigmas de (lenguajes de) programación

2 de septiembre de 2025

# Esquemas de recursión estructural sobre listas

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía, y el tipo del resultado debe ser el de los elementos de la lista.

# Esquemas de recursión estructural sobre listas

Para situaciones en las cuales no hay un caso base claro (ej: no existe el neutro), tenemos las funciones: `foldr1` y `foldl1`. Permiten hacer recursión estructural sobre listas sin definir un caso base:

- `foldr1` toma como caso base el último elemento de la lista.
- `foldl1` toma como caso base el primer elemento de la lista.

Para ambas, la lista **no** debe ser vacía, y el tipo del resultado debe ser el de los elementos de la lista.

## Definir alguna de las siguientes funciones

- `last :: [a] -> a`
- `maximum :: Ord a => [a] -> a`
- `mejorSegún :: (a -> a -> Bool) -> [a] -> a`

# Recursión sobre listas

Implementar las siguientes funciones utilizando esquemas de recursión

- 1 `elem :: Eq a => a -> [a] -> Bool` que indica si un elemento pertenece o no a la lista.
- 2 `take :: Int -> [a] -> [a]` utilizando `foldr`.
- 3 `sacarUna :: Eq a => a -> [a] -> [a]` que elimina la primera aparición de un elemento en la lista.

# Recursión sobre listas

Implementar las siguientes funciones utilizando esquemas de recursión

- 1 `elem :: Eq a => a -> [a] -> Bool` que indica si un elemento pertenece o no a la lista.
- 2 `take :: Int -> [a] -> [a]` utilizando `foldr`.
- 3 `sacarUna :: Eq a => a -> [a] -> [a]` que elimina la primera aparición de un elemento en la lista.

¿Qué otros esquemas de recursión conocen?

# Recursión sobre listas

Implementar las siguientes funciones utilizando esquemas de recursión

- 1 **elem** :: Eq a => a -> [a] -> Bool que indica si un elemento pertenece o no a la lista.
- 2 **take** :: Int -> [a] -> [a] utilizando foldr.
- 3 **sacarUna** :: Eq a => a -> [a] -> [a] que elimina la primera aparición de un elemento en la lista.

¿Qué otros esquemas de recursión conocen?

¿Cuáles de ellos pueden aplicarse a listas infinitas?

# Un breve repaso

¿Qué tipo de recursión tiene cada una de las siguientes funciones? (Estructural, primitiva, global).

```
take' :: [a] -> Int -> [a]
take' [] _ = []
take' (x:xs) n = if n==0 then [] else x:take' xs (n-1)
```

```
listasQueSuman :: Int -> [[Int]]
listasQueSuman 0 = [[]]
listasQueSuman n | n > 0 =
    [x : xs | x <- [1..n], xs <- listasQueSuman (n-x)]
```

```
fact :: Int -> Int
fact 0 = 1
fact n | n > 0 = n * fact (n-1)
```

```
fibonacci :: Int -> Int
fibonacci 0 = 1
fibonacci 1 = 1
fibonacci n | n > 1 = fibonacci (n-1) + fibonacci (n-2)
```

# Generación Infinita

## Definir:

`pares :: [(Int, Int)]`, una lista (infinita) que contenga **todos** los pares de números naturales (sin repetir).



# Generación Infinita

## Definir:

`pares :: [(Int, Int)]`, una lista (infinita) que contenga **todos** los pares de números naturales (sin repetir).

## Práctica 1, Ejercicio 21:

Definir en Haskell una lista que contenga todas las listas finitas de enteros positivos (esto es, con elementos mayores o iguales que 1).

# Folds sobre estructuras nuevas

Sea el siguiente tipo:

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)
```

Ejemplo: `miÁrbol = Bin (Hoja 3) 5 (Bin (Hoja 7) 8 (Hoja 1))`

Definir el esquema de recursión estructural (*fold*) para árboles estrictamente binarios, y dar su tipo.

El esquema debe permitir definir las funciones `altura`, `ramas`, `cantNodos`, `cantHojas`, `espejo`, etc.

# ¿Cómo hacemos?

Recordemos el tipo de `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

(Pista: pensar en cuáles son los constructores del tipo `[a]`).

# ¿Cómo hacemos?

Recordemos el tipo de `foldr`, el esquema de recursión estructural para listas.

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

¿Por qué tiene ese tipo?

(Pista: pensar en cuáles son los constructores del tipo `[a]`).

Un esquema de recursión estructural espera recibir **un argumento por cada constructor** (para saber qué devolver en cada caso), y además **la estructura que va a recorrer**.

El tipo de cada argumento va a depender de lo que reciba el constructor correspondiente. (¡Y todos van a devolver lo mismo!)

Si el constructor es recursivo, el argumento correspondiente del fold va a recibir el resultado de cada llamada recursiva.

# ¿Cómo hacemos? (Continúa)

Miremos bien la estructura del tipo.

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)
```

Estamos ante un tipo inductivo con un constructor *no recursivo* y un constructor *recursivo*.

# ¿Cómo hacemos? (Continúa)

Miremos bien la estructura del tipo.

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)
```

Estamos ante un tipo inductivo con un constructor *no recursivo* y un constructor *recursivo*.

¿Cuál va a ser el tipo de nuestro fold?

# ¿Cómo hacemos? (Continúa)

Miremos bien la estructura del tipo.

```
data AEB a = Hoja a | Bin (AEB a) a (AEB a)
```

Estamos ante un tipo inductivo con un constructor *no recursivo* y un constructor *recursivo*.

¿Cuál va a ser el tipo de nuestro fold?

¿Y la implementación?

# Tipo

- Queremos ir de un AEB a a un b cualquiera.

```
foldAEB ::
```



# Tipo

- Queremos ir de un AEB a a un b cualquiera.

```
foldAEB ::
```

```
      AEB a          -- Nuestro árbol  
-> b                -- Nuestro resultado
```

# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.

```
foldAEB ::
```

```
      AEB a          -- Nuestro árbol  
->    b              -- Nuestro resultado
```

# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.

```
foldAEB ::  
    -                               -- Si es Hoja  
-> -                               -- Si es Bin  
-> AEB a                           -- Nuestro árbol  
-> b                               -- Nuestro resultado
```

# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.
- En cada caso necesitamos devolver un el mismo tipo de resultado: b.

```
foldAEB ::  
    -                               -- Si es Hoja  
->  -                               -- Si es Bin  
->  AEB a                           -- Nuestro árbol  
->  b                               -- Nuestro resultado
```

# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.
- En cada caso necesitamos devolver un el mismo tipo de resultado: b.

```
foldAEB ::  
    (_ -> b)           -- Si es Hoja  
-> (_ -> b)            -- Si es Bin  
-> AEB a               -- Nuestro árbol  
-> b                   -- Nuestro resultado
```

# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.
- En cada caso necesitamos devolver un el mismo tipo de resultado: b.
- Los parámetros de cada constructor aparecen en el mismo orden en las funciones.

```
foldAEB ::
    (_ -> b)           -- Si es Hoja
-> (_ -> b)            -- Si es Bin
-> AEB a               -- Nuestro árbol
-> b                   -- Nuestro resultado
```

# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.
- En cada caso necesitamos devolver un el mismo tipo de resultado: b.
- Los parámetros de cada constructor aparecen en el mismo orden en las funciones.

```
foldAEB ::
    (_ -> b)           -- Si es Hoja
-> (_ -> _ -> _ -> b)  -- Si es Bin
-> AEB a               -- Nuestro árbol
-> b                   -- Nuestro resultado
```

# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.
- En cada caso necesitamos devolver un el mismo tipo de resultado: b.
- Los parámetros de cada constructor aparecen en el mismo orden en las funciones.
- Los parámetros recursivos son un b, el resto mantiene el tipo.

```
foldAEB ::
    (_ -> b)                -- Si es Hoja
->  (_ -> _ -> _ -> b)      -- Si es Bin
->  AEB a                   -- Nuestro árbol
->  b                      -- Nuestro resultado
```



# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.
- En cada caso necesitamos devolver un el mismo tipo de resultado: b.
- Los parámetros de cada constructor aparecen en el mismo orden en las funciones.
- Los parámetros recursivos son un b, el resto mantiene el tipo.

```
foldAEB ::
    (a -> b)           -- Si es Hoja
-> (b -> a -> b -> b)  -- Si es Bin
-> AEB a              -- Nuestro árbol
-> b                  -- Nuestro resultado
```

# Tipo

- Queremos ir de un AEB a a un b cualquiera.
- Necesitamos saber qué hacer para cada constructor.
- En cada caso necesitamos devolver un el mismo tipo de resultado: b.
- Los parámetros de cada constructor aparecen en el mismo orden en las funciones.
- Los parámetros recursivos son un b, el resto mantiene el tipo.
- Listo!

```
foldAEB ::
```

	(a -> b)	-- Si es Hoja
->	(b -> a -> b -> b)	-- Si es Bin
->	AEB a	-- Nuestro árbol
->	b	-- Nuestro resultado

# Solución

```
foldAEB :: (a -> b) -> (b -> a -> b -> b) -> AEB a -> b
foldAEB fHoja fBin t    =    case t of
    Hoja n              ->    fHoja n
    Bin t1 n t2         ->    fBin (rec t1) n (rec t2)
                                where rec = foldAEB fHoja fBin
```

Ejercicio para ustedes: definir las funciones altura, ramas, cantNodos, cantHojas y espejo usando foldAEB.

Si quieren podemos hacer alguna en el pizarrón.

# Funciones sobre árboles

Dado el tipo de datos:

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

¿Qué tipo de recursión tiene cada una de las siguientes funciones?  
(Estructural, primitiva, global).

```
insertarABB :: Ord a => a -> AB a -> AB a
insertarABB x Nil = Bin Nil x Nil
insertarABB x (Bin i r d) = if x < r
    then Bin (insertarABB x i) r d
    else Bin i r (insertarABB x d)
```

```
truncar :: AB a -> Int -> AB a
truncar Nil _ = Nil
truncar (Bin i r d) n = if n == 0 then Nil else
    Bin (truncar i (n-1)) r (truncar d (n-1))
```

Tarea: prueben escribir estas funciones con los esquemas de recursión

# Folds sobre otras estructuras

Dado el siguiente tipo que representa polinomios:

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

- Definir la función

```
evaluar :: Num a => a -> Polinomio a -> a
```

# Folds sobre otras estructuras

Dado el siguiente tipo que representa polinomios:

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

- Definir la función  
evaluar :: Num a => a -> Polinomio a -> a
- Definir el esquema de recursión estructural foldPoli para polinomios (y dar su tipo).

# Folds sobre otras estructuras

Dado el siguiente tipo que representa polinomios:

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

- Definir la función  
`evaluar :: Num a => a -> Polinomio a -> a`
- Definir el esquema de recursión estructural `foldPoli` para polinomios (y dar su tipo).
- Redefinir `evaluar` usando `foldPoli`.

# Una estructura más compleja

Dado el tipo de datos

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

de árboles donde cada nodo tiene una cantidad indeterminada de hijos.

- ❶ Escribir el esquema de recursión estructural para `RoseTree`.
- ❷ Usando el esquema definido, escribir las siguientes funciones:
  - `hojas`, que dado un `RoseTree`, devuelva una lista con sus hojas ordenadas de izquierda a derecha, según su aparición en el `RoseTree`.
  - `ramas`, que dado un `RoseTree`, devuelva los caminos de su raíz a cada una de sus hojas.
  - `tamaño`, que devuelve la cantidad de nodos de un `RoseTree`.
  - `altura`, que devuelve la altura de un `RoseTree` (la cantidad de nodos de la rama más larga). Si el `RoseTree` es una hoja, se considera que su altura es 1.



# Funciones como estructuras de datos

## Representando conjuntos con funciones

Se cuenta con la siguiente representación de conjuntos `type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si  $c$  es un conjunto y  $e$  un elemento, la expresión  $c\ e$  devuelve `True` si  $e$  pertenece a  $c$  y `False` en caso contrario.

1 Definir la constante vacío `:: Conj a`, y las funciones:

- `insertar :: Eq a => a -> Conj a -> Conj a`
- `pertenece :: Eq a => a -> Conj a -> Bool`
- `eliminar :: Eq a => a -> Conj a -> Conj a`

# Funciones como estructuras de datos

## Representando conjuntos con funciones

Se cuenta con la siguiente representación de conjuntos  
`type Conj a = (a->Bool)` caracterizados por su función de pertenencia. De este modo, si  $c$  es un conjunto y  $e$  un elemento, la expresión  $c\ e$  devuelve `True` si  $e$  pertenece a  $c$  y `False` en caso contrario.

- 1 Definir la constante vacío `:: Conj a`, y las funciones:
  - `insertar :: Eq a => a -> Conj a -> Conj a`
  - `pertenece :: Eq a => a -> Conj a -> Bool`
  - `eliminar :: Eq a => a -> Conj a -> Conj a`
- 2 Escribir las funciones intersección, unión y diferencia (todas de tipo `Conj a -> Conj a -> Conj a`).

# Recomendación

Revisar la sección útil del campus, hay presentes apuntes sobre:

- Recursión
- Generación infinita

*i i i i i i i i i i ? ? ? ? ? ? ? ?*