# Programación Funcional en Haskell

Paradigmas de Lenguajes de Programación

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

22 de agosto de 2017

# Hoy presentamos...

1. Esquemas de recursión sobre listas
   - Map
   - Filter
2. Folds sobre listas
   - FoldR
   - FoldL
3. Otros esquemas de recursión sobre listas
4. Tipos algebraicos
5. Recursión estructural en tipos algebraicos

# Esquemas de recursión sobre listas: Map

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

# Esquemas de recursión sobre listas: Map

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

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

# Esquemas de recursión sobre listas: Map

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

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

O, hablando en francés, la función `map`

- Toma una función que sabe como convertir un tipo **a** en otro **b**,
- Y nos devuelve una función que sabe como convertir listas de **a** en listas de **b**.

# Esquemas de recursión sobre listas: Map

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

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

O, hablando en francés, la función `map`

- Toma una función que sabe como convertir un tipo **a** en otro **b**,
- Y nos devuelve una función que sabe como convertir listas de **a** en listas de **b**.

```
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

# Esquemas de recursión sobre listas: Map

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

La función `map` nos permite procesar todos los elementos de una lista mediante una transformación.

O, hablando en francés, la función `map`

- Toma una función que sabe como convertir un tipo **a** en otro **b**,
- Y nos devuelve una función que sabe como convertir listas de **a** en listas de **b**.

```
map f [] = []
map f (x:xs) = (f x):(map f xs)
```

## Definir utilizando map

- `longitudes :: [[a]] -> [Int]`
- `losIesimos :: [Int] -> [[a] -> a]` que devuelve una lista con las funciones que toman los iésimos de una lista.
- `shuffle :: [Int] -> [a] -> [a]` que, dada una lista de índices $[i_1, \ldots, i_n]$ y una lista $l$, devuelve la lista $[l_{i_1}, \ldots, l_{i_n}]$

# Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

# Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

# Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

O, hablando en francés, la función `filter`

- Toma una función que nos dice si un elemento cumple una condicón,
- Y nos devuelve una función que sabe como convertir listas de elementos cualquiera en listas cuyos elementos cumplen la condición deseada.

# Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

O, hablando en francés, la función `filter`

- Toma una función que nos dice si un elemento cumple una condicón,
- Y nos devuelve una función que sabe como convertir listas de elementos cualquiera en listas cuyos elementos cumplen la condición deseada.

```
filter p [] = []
filter p (x:xs) = if p x then x:(filter xs) else filter xs
```

# Esquemas de recursión sobre listas: Filter

```
filter :: (a -> Bool) -> [a] -> [a]
```

La función `filter` nos permite obtener los elementos de una lista que cumplen cierta condición.

O, hablando en francés, la función `filter`

- Toma una función que nos dice si un elemento cumple una condicón,
- Y nos devuelve una función que sabe como convertir listas de elementos cualquiera en listas cuyos elementos cumplen la condición deseada.

```
filter p [] = []
filter p (x:xs) = if p x then x:(filter xs) else filter xs
```

## Definir utilizando filter

- `deLongitudN :: Int -> [[a]] -> [[a]]`
- `soloPuntosFijos :: [Int -> Int] -> Int -> [Int -> Int]` que toma una lista de funciones y un número *n*. En el resultado, deja las funciones que al aplicarlas a *n* dan *n*.
- `quickSort :: Ord a => [a] -> [a]`

# Esquemas de recursión sobre listas: FoldR

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

# Esquemas de recursión sobre listas: FoldR

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

# Esquemas de recursión sobre listas: FoldR

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, hablando en francés, la función `foldr`

- Toma una función que representa el paso recursivo y un valor que representa el caso base,
- Y nos devuelve una función que sabe como reducir listas de **a** a un valor **b**.

# Esquemas de recursión sobre listas: FoldR

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, hablando en francés, la función `foldr`

- Toma una función que representa el paso recursivo y un valor que representa el caso base,
- Y nos devuelve una función que sabe como reducir listas de **a** a un valor **b**.

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

# Esquemas de recursión sobre listas: FoldR

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

La función `foldr` nos permite realizar recursión estructural sobre una lista.

O, hablando en francés, la función `foldr`

- Toma una función que representa el paso recursivo y un valor que representa el caso base,
- Y nos devuelve una función que sabe como reducir listas de **a** a un valor **b**.

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

## Definir utilizando foldr

- `longitud :: [a] -> Int`
- `concatenar :: [[a]] -> [a]`
- `suma :: [Int] -> Int`

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
---> 1 + 5
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
---> 1 + 5
---> 6
```

# Esquemas de recursión sobre listas: FoldR

## FoldR

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldr (+) 0 [1,2,3]
---> 1 + (foldr (+) 0 [2,3])
---> 1 + (2 + (foldr (+) 0 [3]))
---> 1 + (2 + (3 + (foldr (+) 0 [])))
---> 1 + (2 + (3 + 0))
---> 1 + (2 + 3)
---> 1 + 5
---> 6
```

Notar que el primer `(+)` que se puede resolver es entre el último elemento de la lista y el caso base del `foldr`. Por esta razón decimos que el `foldr` *acumula* el resultado desde la **derecha**.

# Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

# Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

## FoldL

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

# Esquemas de recursión sobre listas: FoldL

La función `foldl` es muy similar a `foldr` pero *acumula* desde la **izquierda**.
Se define de la siguiente forma:

## FoldL

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

## Definir utilizando foldl

- `reverso :: [a] -> [a]`
- `suma :: [Int] -> Int`

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
---> (3 + 3)
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
---> (3 + 3)
---> 6
```

# Esquemas de recursión sobre listas: FoldL

## FoldL

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

## ¿Cómo funciona?

```
suma [1,2,3]
---> foldl (+) 0 [1,2,3]
---> foldl (+) (0 + 1) [2,3]
---> foldl (+) ((0 + 1) + 2) [3]
---> foldl (+) (((0 + 1) + 2) + 3) []
---> (((0 + 1) + 2) + 3)
---> ((1 + 2) + 3)
---> (3 + 3)
---> 6
```

Notar que el primer `(+)` que se puede resolver es entre el primer elemento de la lista y el caso base del `foldl`.

# Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

## Usando foldr

```
suma [1..]
```

## Usando foldl

```
suma [1..]
```

# Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

## Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
```

## Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
```

# Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

## Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
```

## Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
```

# Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

## Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
---> 1 + (2 + (foldr (+) 0 [3..]))
```

## Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
---> foldl (+) ((0 + 1) + 2) [3..]
```

# Esquemas de recursión sobre listas: FoldR, FoldL y las listas infinitas

¿Qué sucede con las listas infinitas al usar `foldr` o `foldl`?

## Usando foldr

```
suma [1..]
---> foldr (+) 0 [1..]
---> 1 + (foldr (+) 0 [2..])
---> 1 + (2 + (foldr (+) 0 [3..]))
---> 1 + (2 + (3 + (foldr (+) 0 [4..])))
```

## Usando foldl

```
suma [1..]
---> foldl (+) 0 [1..]
---> foldl (+) (0 + 1) [2..]
---> foldl (+) ((0 + 1) + 2) [3..]
---> foldl (+) (((0 + 1) + 2) + 3) [4..]
```

# Esquemas de recursión sobre listas: FoldR1 y FoldL1

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.

# Esquemas de recursión sobre listas: FoldR1 y FoldL1

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.

## Definir las siguientes funciones

- `ultimo :: [a] -> a`
- `maximum :: Ord a => [a] -> a`

# Esquemas de recursión sobre listas: FoldR1 y FoldL1

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.

## Definir las siguientes funciones

- `ultimo :: [a] -> a`
- `maximum :: Ord a => [a] -> a`

## ¿Qué computan estas funciones?

- ```
  f1 :: [Bool] -> Bool
  f1 = foldr (&&) True
  ```
- ```
  f2 :: [a] -> [a]
  f2 = foldr (:) []
  ```
- ```
  f3 :: [a] -> [a] -> [a]
  f3 xs ys = foldr (:) ys xs
  ```
- ```
  f4 :: [a] -> [a]
  f4 = foldl (flip (:)) []
  ```

# ¡Las difíciles!

**Calentando motores... (no vale recursión explícita)**

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece e = foldr ...
```

# ¡Las difíciles!

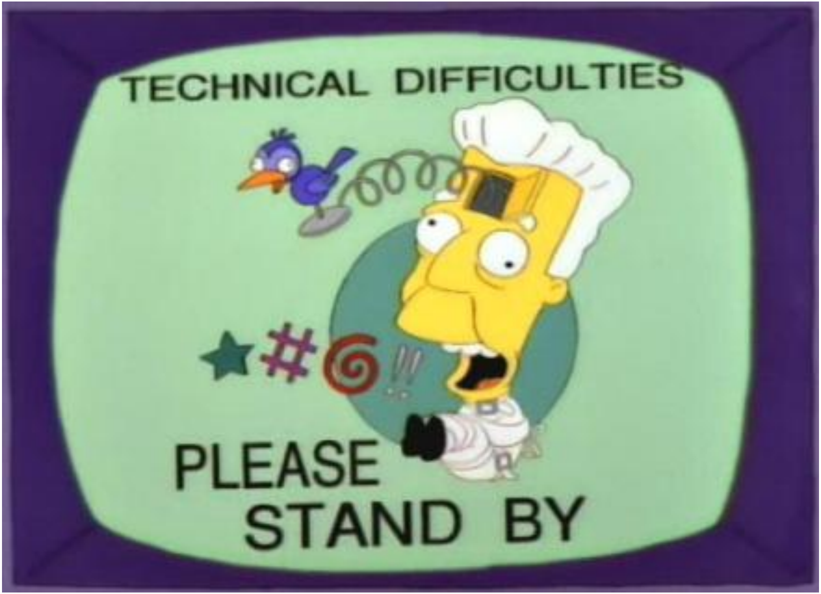
## Calentando motores... (no vale recursión explícita)

```
pertenece :: Eq a => a -> [a] -> Bool
pertenece e = foldr ...
```

## Definir la función take, ¿cuál es la diferencia?

```
take :: Int -> [a] -> [a]
take n = foldr ...
```

# Break



# Otros esquemas de recursión sobre listas: Divide & Conquer

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b
= (a -> Bool)              – determina si es o no el caso trivial
-> (a -> b)                – resuelve el caso trivial
-> (a -> [a])              – parte el problema en sub-problemas
-> ([b] -> b)              – combina resultados
-> a                       – input
-> b                       – resultado
```

# Otros esquemas de recursión sobre listas: Divide & Conquer

La técnica de Divide & Conquer consiste en dividir un problema en problemas más fáciles de resolver y luego combinando los resultados parciales, lograr obtener un resultado general.

Para generalizar la técnica, crearemos el tipo `DivideConquer` definido como:

```
type DivideConquer a b
= (a -> Bool)        -- determina si es o no el caso trivial
-> (a -> b)          -- resuelve el caso trivial
-> (a -> [a])        -- parte el problema en sub-problemas
-> ([b] -> b)        -- combina resultados
-> a                 -- input
-> b                 -- resultado
```

## Definir las siguientes funciones

- ```
  dc :: DivideConquer a b
  dc esTrivial resolver repartir combinar x = ...
  ```
- ```
  mergeSort :: Ord a => [a] -> [a]
  mergeSort = dc ...
  ```

# Tipos algebraicos y su definición en Haskell

## Tipos algebraicos

- definidos como **combinación de otros tipos**
- están formados por uno o más constructores
- cada constructor puede o no tener argumentos
- los argumentos de los constructores pueden ser recursivos
- se inspeccionan usando *pattern matching*
- se definen mediante la cláusula `data`

## Algunos ejemplos

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
```

# Tipos algebraicos y su definición en Haskell

## Tipos algebraicos

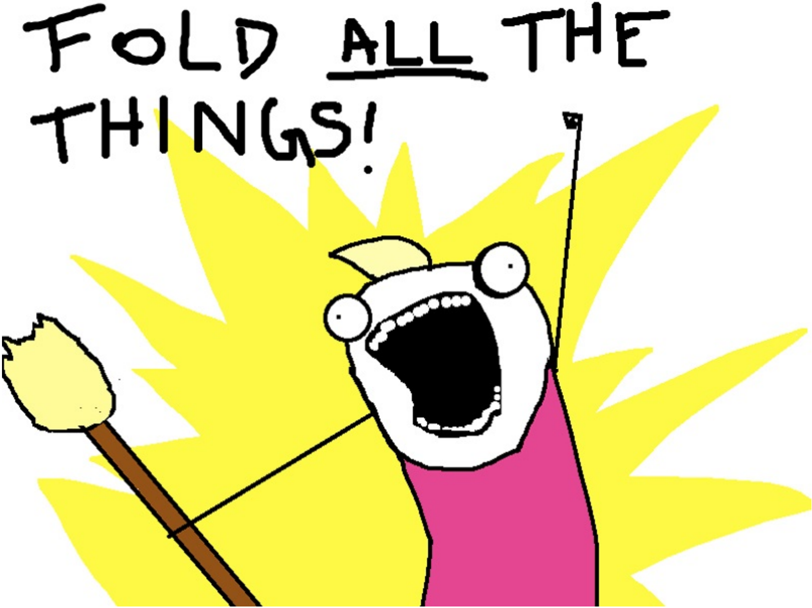
- definidos como **combinación de otros tipos**
- están formados por uno o más constructores
- cada constructor puede o no tener argumentos
- los argumentos de los constructores pueden ser recursivos
- se inspeccionan usando *pattern matching*
- se definen mediante la cláusula `data`

## Algunos ejemplos

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b

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

## Folds sobre estructuras nuevas



## ¿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.

# Folds sobre estructuras nuevas

Definir el esquema de recursión estructural para el siguiente tipo:

```
data Formula = Proposicion String | No Formula
                                  | Y Formula Formula
                                  | O Formula Formula
                                  | Imp Formula Formula
```

# Folds sobre estructuras nuevas

Definir el esquema de recursión estructural para el siguiente tipo:

```
data Formula = Proposicion String | No Formula
                                  | Y Formula Formula
                                  | O Formula Formula
                                  | Imp Formula Formula
```

## Ejercicio

Usando el esquema definido, escribir las funciones:

- `proposiciones :: Formula -> [String]`
- `quitarImplicaciones :: Formula -> Formula` que convierte todas las formulas de la pinta $(p \implies q)$ a $(\neg p \vee q)$
- `evaluar :: [(Proposicion, Bool)] -> Formula -> Bool` que dada una formula y los valores de verdad asignados a cada una de sus proposiciones, nos devuelve el resultado de evaluar la fórmula lógica.

¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿? ¿?