

## Programación Funcional en Haskell (cont.)

Paradigmas de Lenguajes de Programación

# Esquemas sobre listas

## 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

## Esquema

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

`foldr` :  $g == \text{foldr } f \ z$

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

# Repensamos algunas funciones

## Las del inicio

```
sumaL    = foldr (+) 0
concat    = foldr (++) []
reverse   = foldr ((flip (++) ) . (:[])) []
```

## Map y Filter

```
map f      = foldr ((:) . f ) []
filter p    = foldr (\x xs -> if p x then x:xs else xs) []
```

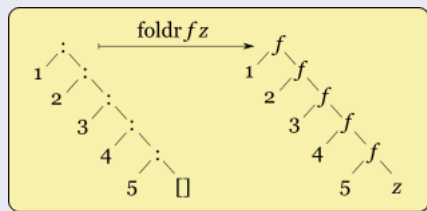
# Sobre foldr

¿Qué función estamos definiendo?

??? :: ???

??? = **foldr** (:) []

foldr f



Definición de `(++)`

`(++) :: [a] -> [a] -> [a]`

`xs ++ ys = foldr (:) ys xs`

Longitud y suma con una sola pasada sobre la lista

```
sumaLong :: [Int] -> (Int, Int)
sumaLong = foldr (\x (s,l) -> (s+x, l+1)) (0,0)
```

## dropWhile

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

## dropWhile'

```
dropWhile' :: (a -> Bool) -> [a] -> ([a],[a])
dropWhile' p = foldr f ([],[])
  where f x (ys, xs) = (if p x then ys else x:xs, x:xs)
```

# Recursión Primitiva

## Esquema

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

`recr : g == recr z f`

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



dropWhile

foldr en términos de recr

recr en términos de foldr

# foldl

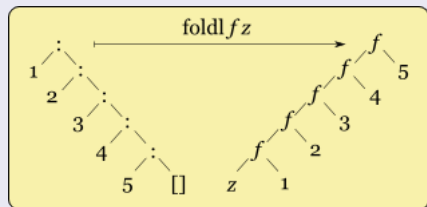
## foldl

`foldl1 :: (b -> a -> b) -> b -> [a] -> b`

`foldl1 _ z [] = z`

`foldl1 f z (x:xs) = foldl1 f (f z x) xs`

## foldr f



## Repensamos reverse

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

## Para pensar : foldr vs foldl

- en listas infinitas...
- foldl en términos de foldr
- y foldr en términos de foldl?

# Esquemas sobre tipos de datos algebraicos

## Tipo árboles binarios

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

## Map en árboles

```
mapA :: (a -> b) -> Arbol a -> Arbol b
mapA f (Hoja x) = Hoja (f x)
mapA f (Nodo x i d) = Nodo (f x) (mapA f i) (mapA f d)
```

## Fold en árboles

```
foldA :: (a -> b) -> (a -> b -> b -> b) -> Arbol a -> b
foldA f g (Hoja x) = f x
foldA f g (Nodo x i d) = g x (foldA f g i) (foldA f g d)
```

## Tipo árboles

```
data Arbol a = Hoja a | Nodo a (Arbol a) (Arbol a)
```

## Fold en árboles

```
foldA :: (a -> b) -> (a -> b -> b -> b) -> Arbol a -> b
```

## Tipo de los constructores

```
Hoja :: a -> Arbol a
```

```
Nodo :: a -> Arbol a -> Arbol a -> Arbol a
```

## Identidad en árboles

```
foldA Hoja Nodo
```

# Esquemas sobre tipos de datos algebraicos

## Tipo árboles generales

```
data AG a = NodoAG a [AG a]
```

## Map en árboles

```
mapAG :: (a -> b) -> AG a -> AG b  
mapAG f (NodoAG a as) = NodoAG (f a) (map (mapAG f) as)
```

## Fold en árboles

```
foldAG :: (a -> [b] -> b) -> AG a -> b  
foldAG f (NodoAG a as) = f a (map (foldAG f) as)
```

Hutton, G. (1999). A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4), 355-372.