# Programación Funcional

Curso 2019-20

## ORDEN SUPERIOR

# Funciones de orden superior

*(HO functions)*

- Funciones con algún argumento (o el resultado) de tipo funcional.
- Favorecen la abstracción, concisión y reutilización de código.

Funciones de primer orden: las "normales", que no son de OS

`f ::(Int->Int)->Int` Es de orden superior

`f' ::(Int,Int)->Int` Es de primer orden

¿Cuál es el orden de la función g definida como `g x = f'`

## Un par de funciones de primer orden...

```
-- incList xs:  sumar uno a todos los elementos de xs
-- incList [x1,...,xn] = [1+x1,...,1+xn]
incList::[Int] -> [Int]
incList [] = []
incList (x:xs) = inc x:incList xs where inc x = x+1
```

Recomendado: declarar siempre los tipos

```
-- lengthList xss: longitudes de los elementos de xss
-- lengthList [xs1,...,xsn] = [length xs1,...,length xsn]
lengthList::[[a]] -> [Int]
lengthList []     = []
lengthList (x:xs) = length x:lengthList xs
```

Se repite la misma estructura:
aplicar una función $f$ a todos los elementos de la lista

## Abstracción de OS: `map`

```
-- map f xs = resultado de aplicar f a todos los elementos de xs
-- map f [x1,...,xn] = [f x1,...,f xn]
map::(a -> b) -> [a] -> [b]
map f []     = []
map f (x:xs) = f x:map f xs
```

```
> map (take 2) [[1,2,3],[2],[7,6,4]]
[[1,2],[2],[7,6]]
> map not [True,False,False]
[False,True,True]
```

```
-- incList [x1,...,xn] = [1+x1,...,1+xn]
-- lengthList [xs1,...,xsn] = [length xs1,...,length xsn]
incList xs    = map inc xs
lengthList xs = map length xs
```

## Cancelación de argumentos

### En lugar de

```
incList xs    = map inc xs
lengthList xs = map length xs
```

### Podemos definir de modo equivalente

```
incList' = map inc
lengthList' = map length
```

*Pointless programming*

Tanto recursión como aplicación a argumentos quedan implícitas

### Ambas definiciones son equivalentes

```
incList' [1,2,3] = (map inc) [1,2,3] = map inc [1,2,3] =
incList [1,2,3]
lengthList' [1,2,3] = (map length) [1,2,3] =
map length [1,2,3] = lengthList [1,2,3]
```

# Funciones de orden superior sobre listas I

## filter, all, any

```
-- filter p xs = lista de elementos de xs que cumplen la propiedad p
filter::(a -> Bool) -> [a] -> [a]
filter p []    = []
filter p (x:xs)
  | p x        = x:filter p xs
  | otherwise =   filter p xs


-- all p xs = todos los elementos de xs cumplen p
-- any p xs = algún elemento de xs cumple p

all _ []          =  True
all p (x:xs)      =  p x && all p xs
any _ []          = False
any p (x:xs)      = p x || any p xs
```

```
> filter (> 1) [-1,3,0,4]
[3,4]
> all (> 1) [-1,3,0,4]
False
> any (> 1) [-1,3,0,4]
True
```

# Funciones de orden superior sobre listas II

## takeWhile, dropWhile, span, break

```
-- takeWhile p xs = mayor prefijo de xs cuyos elementos cumplen p
-- dropWhile p xs = resultado de eliminar (takeWhile p xs) de xs
takeWhile, dropWhile:: (a -> Bool) -> [a] -> [a]
span, break :: (a -> Bool) -> [a] -> ([a], [a])
takeWhile _ []   = []
takeWhile p (x:xs)
     | p x       = x : takeWhile p xs
     | otherwise = []

dropWhile _ []   = []
dropWhile p (x:xs)
     | p x       = dropWhile p xs
     | otherwise = x:xs

span p  xs = (takeWhile p xs,dropWhile p xs)
break p xs = span (not.p) xs
```

- . es la composición de funciones xs se podría cancelar, pero p no

## Composición de funciones: .

```
-- f.g = composición de las funciones f y g

infixr 9 .
(.):: (b -> c) -> (a -> b) -> (a -> c)
(.) f g x = f (g x)
```

```
> (head .  tail) [1,2,3]
2
```

```
> ((^ 2).(3 *)) 2
36
```

## Aplicación funcional: $

```
-- f $ x = aplica la función f a x

infixr 0 $
($) :: (a -> b) -> a -> b
($) f x = f (x)
```

```
> head $ tail [1,2,3]
2
```

```
> tail $ map (*2) [1,2,3]
[4,6]
```

# Más funciones de orden superior sobre listas

**iterate, zipWith, zip**

```
-- iterate f x = [x,f x, f (f x),...]
iterate:: (a -> a) -> a -> [a]
iterate f x = x : iterate f (f x)
```

```
iterate (* 2) 1 =[1,2,4,8,...]
```

```
take 3 (map head (iterate tail [1..]))=[1,2,3]
```

```
-- zipWith f xs ys:  combina los elementos de xs e ys mediante f
-- Si una lista es más corta, se descartan los sobrantes de la otra
-- zipWith f [x1,...,xn] [y1,...,ym]=[f x1 y1,...,f xk yk] (k=min(n,m))

zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _      _      = []
```

```
zip [1,2,3] ['a','b'] = [(1,'a'),(2,'b')]
```

```
zipWith (+) [1,2,3] [4,5,6] = [5,7,9]
```

Define la función `zip` en términos de `zipWith`

# La familia *fold*

## Un patrón de recursión sobre listas muy repetido

```
sum [] = 0
sum (x:xs) = x + sum xs
```

```
and [] = True
and (x:xs) = x && and xs
```

```
or [] = False
or (x:xs) = x || or xs
```

```
length [] = 0
length (x:xs) = 1+length xs
```

```
concat [] = []
concat (xs:xss) = xs++concat xss
```

El patrón general que reconocemos es:

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

O bien, si f viene como un operador infijo $\bigoplus$:

$$
\begin{array}{lll}
\texttt{g} & \texttt{[]} & = \texttt{e} \\
\texttt{g} & \texttt{(x:xs)} & = \texttt{x} \bigoplus \texttt{g xs}
\end{array}
$$

O, en términos semiformales, y suponiendo que $\bigoplus$ asocia a la derecha:

$$\texttt{g [x1,x2,...,xn]} = \texttt{x1} \bigoplus \texttt{x2} \bigoplus \ldots \bigoplus \texttt{xn} \bigoplus \texttt{e}$$

Podemos abstraer en una función de OS con f/$\bigoplus$ y e como parámetros

# La familia *fold* (II)

foldr f e [x1,x2,...,xn] = f x1 (f x2 (. . . (f xn e). . . ))
foldr $\bigoplus$ e [x1,x2,...,xn] = x1 $\bigoplus$ x2 $\bigoplus$ . . . $\bigoplus$ xn $\bigoplus$ e

```
foldr:: (a -> b -> b) -> b -> [a] -> b
foldr f e []     = e
foldr f e (x:xs) = f x (foldr f e xs)
-- O escribiendo f en modo infijo
foldr f e (x:xs) = x `f` (foldr f e xs)
```

```
sum     = foldr (+) 0
product = foldr (*) 1
and     = foldr (&&) True
or      = foldr (||) False
length  = foldr f 0  where f x y = y+1
concat  = foldr (++) []
```

# La familia *fold* (III)

foldl $\bigoplus$ e [x1,x2,...,xn] $=$ e $\bigoplus$ x1 $\bigoplus$ x2 $\bigoplus$ $\ldots$ $\bigoplus$ xn

*(suponiendo que $\bigoplus$ asocia por la izda)*

foldl f e [x1,x2,...,xn] $=$ (f (...(f (f e x1) x2)...) xn

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

```
sum     = foldl (+) 0
product = foldl (*) 1
and     = foldl (&&) True
or      = foldl (||) False
length  = foldl f 0
    where f x y = x+1
concat  = foldl (++) []
```

Más ineficiente que antes ¿Por qué?

# La familia *fold* (III)

- *foldr* puede procesar listas (incluso infinitas) sin recorrerlas enteras (dependiendo de $\bigoplus$)
- *foldl* ha de recorrer la lista entera
- *foldl* presenta recursión final (*tail recursion)*
- La eficiencia comparativa es muy dependiente de cada caso

## Miscelánea OS

```
-- flip f: cambia de orden los argumentos de f
flip:: (a -> b -> c) -> (b -> a -> c)
flip f x y = f y x


-- curry f:  currifica f (que es una función que se aplica a pares)
-- uncurry f:  hace que f (función currificada) se aplique a pares
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
```

## Funciones que suelen usarse en forma de aplicación parcial

```
-- id:  función identidad
id::  a -> a
id x = x
-- const x:  función constante de valor x
const::  a -> b -> a
const x y = x
```