

Estructuras de Datos

Grados en Ingeniería Informática, del Software y de Computadores

ETSI Informática

Universidad de Málaga

## **2. Características de la Programación Funcional**

---

José E. Gallardo, Francisco Gutiérrez, Pablo López

Dpto. Lenguajes y Ciencias de la Computación

Universidad de Málaga

# Índice

- Características de la Programación Funcional
  - Listas
  - Patrones
  - Eficiencia. Parámetros Acumuladores
  - Funciones de orden superior: map y filter
  - Secuencias Aritméticas y Listas por Comprensión. Quick Sort.
  - Pruebas con QuickCheck (cont.)
  - Plegado de listas
  - Parcialización. Composición de Funciones
  - Inducción sobre Listas
  - Tipos Algebraicos y clases: Eq, Ord, Show, deriving. Enumerados: booleanos; tipo unión, tipo producto.

# Listas

- Una lista es una secuencia de datos homogéneos
  - Secuencia de Datos: Una lista mantiene varios valores
  - Homogéneos: todos los valores en la lista deben tener el mismo tipo

- Sintaxis:

[]

la lista vacía

[7, 2, 5]

tiene tipo [Integer]

[True, False]

tiene tipo [Bool]

[('a', True), ('a', False)]

tiene tipo [(Char, Bool)]

[7, 'a', 5]



Expresión errónea:

Los elementos deben tener un tipo común



# Operaciones básicas con Listas

- Primer elemento:

Prelude head :: [a] -> a

- Lista sin el primer elemento:

Prelude tail :: [a] -> [a]

- ¿Es vacía? :

Prelude null :: [a] -> Bool

Estas operaciones básicas tienen complejidad O(1)

```
Prelude> head [10,7,3,5]
10
Prelude> tail [True,False,True]
[False,True]
Prelude> null [7,3]
False
Prelude> null []
True
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> tail []
*** Exception: Prelude.tail: empty list
```

No definida para las listas vacías

# Operaciones básicas con Listas (II)

- Podemos añadir un elemento al principio de una lista con el constructor `(::)`

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

Se lee `cons`

- La lista original no se modifica
- Se obtiene una **nueva** lista con un elemento más
- Añadir un elemento al principio tiene complejidad **O(1)**

Una lista se construye  
usando `(::)` y `[]`

`(::)` asocia a la  
derecha

Dos posibles notaciones  
para describir una misma  
lista

```
Prelude> 1 : []
[1]
Prelude> 1 : [3,4,5]
[1,3,4,5]
Prelude> 1 : (2 : (3 : []))
[1,2,3]
Prelude> 1 : 2 : 3 : []
[1,2,3]
Prelude> 1 : 2 : 3 : [] == [1,2,3]
True
```

# Strings

- En Haskell, un String es una lista de caracteres:

Prelude> **type String = [Char]**  Predefinido  
Sinónimo de tipo

- Además de la sintaxis ya vista para expresar listas, las listas de caracteres pueden escribirse entre comillas:

```
Prelude> ['p','e','p','e'] == "pepe"
True
```

```
Prelude> head "pepe"
'p'
```

```
Prelude> tail "pepe"
"epe"
```

```
Prelude> 'p' : "epe"
"pepe"
```

# Definiendo Funciones sobre Listas. Patrones

- Los **Patrones** pueden usarse para definir diferentes casos dependiendo de la forma del argumento
- Los Patrones actúan al contrario que los constructores:
  - Los Constructores componen datos
  - Los Patrones descomponen datos
- Hay distintos patrones para listas:
  - $[]$ ,  $[x]$ ,  $[x, y]$ , ... El argumento debe ser una lista con **exactamente** 0, 1, 2, ... elementos.
    - x se refiere al primer elemento de la lista, y se refiere al segundo, ...
  - $(x:xs)$ ,  $(x:y:xs)$ , ... el argumento debe ser una lista con **al menos** 1, 2, ... elementos.
    - x se refiere a el primer elemento de la lista, y se refiere al segundo, ...
    - xs se refiere al resto de la lista

# Definiendo Funciones sobre Listas. Patrones(II)

- ¿Cuantos elementos tiene una lista?

Prelude length :: [a] -> Int

length [] = 0

length (x:xs) = 1 + length xs

length es una función recursiva

Este patrón descompone la lista en cabeza y cola

- La primera ecuación se usa si la lista es vacía (patrón [])
- La segunda se usa cuando la lista tiene al menos un elemento (patrón (x:xs))
  - x se refiere al primer elemento (**head**) de la lista
  - xs se refiere al resto (**tail**) de la lista

# ¿Cuál es la eficiencia de length?

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

- Para listas con 3 elementos, length realiza 7 pasos de evaluación (reducciones)
- Para listas con 2 elementos, length realiza 5 pasos de evaluación
- Para listas con  $n$  elementos, length realiza  $2n+1$  pasos de evaluación

```
length [1,2,3]
=> {- 2nd equation -}
  1 + length [2,3]
=> {- 2nd equation -}
  1 + (1 + length [3])
=> {- 2nd equation -}
  1 + (1 + (1 + length []))
=> {- 1st equation. Base case -}
  1 + (1 + (1 + 0))
=>
  1 + (1 + 1)
=>
  1 + 2
=>
  3
```

# ¿Cuál es la eficiencia de length? (II)

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

- Sea  $T(n)$  el número de pasos que realiza `length` con una lista de  $n$  elementos:

$$\begin{aligned} T(0) &= 1 \\ T(n) &= 2 + T(n - 1), \text{ si } n > 0 \end{aligned}$$

Un paso para aplicar la primera ecuación

Un paso para aplicar la segunda ecuación y otro para calcular la suma.  $T(n - 1)$  para la recursión

- Solución a la recurrencia:

$$T(n) = 2n + 1$$

- `length` es  **$O(n)$**  (lineal en el número de elementos de la lista)

# Definiendo Funciones sobre Listas. Patrones (III)

- La siguiente función comprueba si una lista está ordenada ascendenteamente:

```
sorted :: (Ord a) => [a] -> Bool
```

```
sorted [] = True
```

```
sorted [_] = True
```

```
sorted (x:y:zs) = x<=y && sorted (y:zs)
```

Tres casos:  
listas con cero, uno y al menos dos elementos

Main> sorted [1,3,4]  
True

Prelude> sorted [7,3,4]  
False

- `[_]` se refiere a lista con un elemento (sin importar qué elemento es)
- `(x:y:zs)` se refiere a listas con al menos dos elementos
  - `x` se refiere a la cabeza de la lista
  - `y` se refiere al segundo elemento
  - `zs` denota el resto de la lista argumento (sin los dos primeros elementos)

# Concatenación de Listas

Prelude> infixr 5 ++  
 $(++) :: [a] \rightarrow [a] \rightarrow [a]$   
 $\boxed{\quad} \quad ++ ys = ys$   
 $(x:xs) ++ ys = x : (xs ++ ys)$

La primera ecuación se usa cuando el primer argumento es la lista vacía. La segunda cuando el primer argumento no es la lista vacía

El segundo argumento es el patrón más general. Puede ser cualquier lista

```
Prelude> [3,4] ++ [1,2,3]
[3,4,1,2,3]
Prelude> [] ++ [3,4,5]
[3,4,5]
Prelude> [3,4,5] ++ []
[3,4,5]
```

- Propiedades de la concatenación de listas (pueden probarse usando inducción):
  - $\boxed{\quad}$  es la identidad para la operación  $(++)$ :
$$[] ++ xs = xs ++ [] = xs$$
  - $(++)$  es asociativa:
$$(xs ++ ys) ++ zs = xs ++ (ys ++ zs)$$

# Concatenación de Listas (II)

- ¿Cuántos pasos de evaluación requiere la concatenación de listas?

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[]      ++ ys   = ys
(x:xs) ++ ys  = x : (xs ++ ys)
```

```
[1,2] ++ [3,4,5]
=> {- 2nd equation -}
    1 : ([2] ++ [3,4,5])
=> {- 2nd equation -}
    1 : (2 : ([] ++ [3,4,5]))
=> {- 1st equation -}
    1 : (2 : [3,4,5])
```

- Si la primera lista tiene 2 elementos, la concatenación lleva 3 pasos de evaluación
- La eficiencia no depende de la segunda lista

# Concatenación de Listas (III)

- ¿Cuántos pasos de evaluación requiere la concatenación de listas?

```
infixr 5 ++
(++) :: [a] -> [a] -> [a]
[] ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- Sea  $T(n)$  el número de pasos de evaluación para computar la concatenación cuando la longitud de la primera lista es  $n$ :

$$\begin{aligned} T(0) &= 1 \\ T(n) &= 1 + T(n - 1), \text{ si } n > 0 \end{aligned}$$

Un paso para aplicar la primera ecuación

Un paso para aplicar la segunda ecuación. : ya está reducido

- Solución a la recurrencia:

$$T(n) = n + 1$$

- La concatenación toma  $O(n)$  (lineal en el número de elementos de la primera lista)

# Invertiendo una Lista. Solución lenta

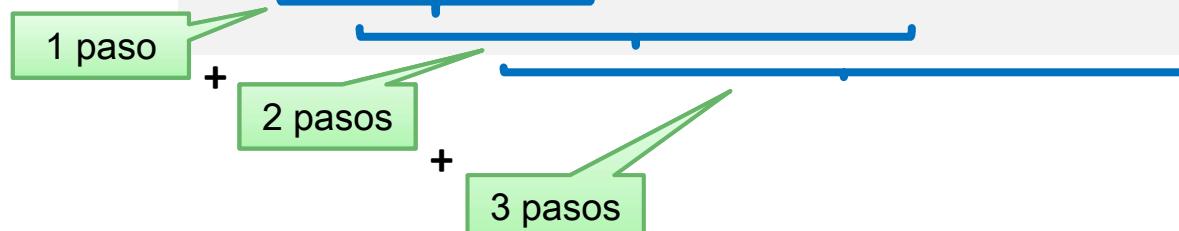
- Para listas con  $n$  elementos, reverse toma  
 $1 + 2 + \dots + (n-1) + n = O(n^2)$  pasos

Prelude

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

- Esta definición de reverse es cuadrática en la longitud de la lista

```
reverse [1,2,3]
=> {- 2nd equation -}
reverse [2,3] ++ [1]
=> {- 2nd equation -}
(reverse [3] ++ [2]) ++ [1]
=> {- 2nd equation -}
((reverse [] ++ [3]) ++ [2]) ++ [1]
=> {- 1st equation -}
(([] ++ [3]) ++ [2]) ++ [1] => ... => [3,2,1]
```



# Invertiendo una Lista. Solución Lenta (II)

Prelude

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

- Sea  $T(n)$  el número de pasos de evaluación para invertir una lista de  $n$  elementos:

$$T(0) = 1$$

Un paso para aplicar la primera  
ecuación

$$T(n) = 1 + O(n) + T(n - 1), \text{ si } n > 0$$

Un paso para aplicar la segunda  
ecuación.  $O(n)$  pasos para la  
concatenación

- Solución a la recurrencia:

$$T(n) = O(n^2)$$

- Esta versión de `reverse` es  $O(n^2)$  (cuadrática en el  
número de elementos de la lista) ☹

# Invertiendo una Lista. Solución rápida

```
Prelude> reverse :: [a] -> [a]
Prelude> reverse xs = revOn xs []
Prelude> where
Prelude>     revOn [] ys = ys
Prelude>     revOn (x:xs) ys = revOn xs (x:ys)
```

revOn xs ys devuelve la lista obtenida al colocar la inversa de la lista xs delante de la lista ys

revOn [1,2,3] [4,5] => [3,2,1,4,5]

- Para una lista de n elementos, reverse toma  $n+2 = O(n)$  pasos
- Esta definición de reverse es **lineal** en la longitud de la lista 😊

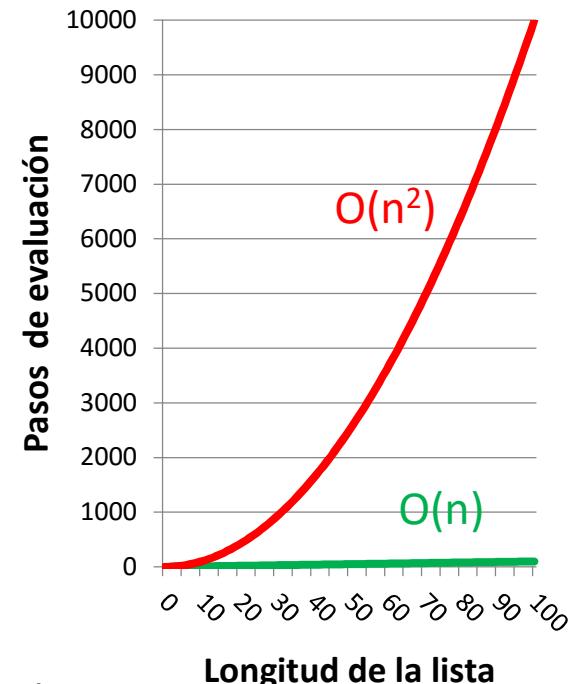
```
reverse [1,2,3]
=> {- reverse definition -}
  revOn [1,2,3] []
=> {- revOn 2nd equation -}
  revOn [2,3] (1 : [])
=> {- revOn 2nd equation -}
  revOn [3] (2 : 1 : [])
=> {- revOn 2nd equation -}
  revOn [] (3 : 2 : 1 : [])
=> {- revOn 1st equation -}
  3 : 2 : 1 : []
```

Solo  $n+2$  pasos de evaluación 😊

# Parámetros Acumuladores

```
reverse :: [a] -> [a]
reverse xs = rev0n xs []
where
    rev0n [] ys = ys
    rev0n (x:xs) ys = rev0n xs (x:ys)
```

- Para listas con  $n$  elementos, reverse toma  $O(n)$  pasos
- Hemos pasado de una definición cuadrática a una lineal 😊
- ¿Cómo lo hemos conseguido?
  - Hemos definido una función auxiliar con un parámetro extra
  - Hemos usado el parámetro extra para calcular el resultado eficientemente
  - Nótese que primero resolvemos eficientemente un problema más general (rev0n), y entonces usamos la solución para resolver nuestro problema original (reverse)
- Esta técnica general se conoce como **parámetros acumuladores**



# Parámetros Acumuladores vs Bucles

- Las definiciones que usan parámetros acumuladores se parecen a los **bucles** en los lenguajes imperativos:

```
factorial :: Integer -> Integer
factorial x = aux 1 x
where
  aux ac 0      = ac
  aux ac n | n>0 = aux (ac*n) (n-1)
```

Inicialización de variables

El acumulador actúa como una variable mutable que es usada para computar el producto

```
int factorial(int x) {
  int n, ac=1;
  for(n=x; n>0; n--)
    ac = ac*n;
  return ac;
}
```

El segundo argumento actúa como el índice del bucle

```
factorial 3
=> {- factorial definition -}
aux 1 3 => aux (1*3) (3-1)
=> {- aux 2nd equation -}
aux (1*3) 2 => aux (1*3*2) (2-1)
=> {- aux 2nd equation -}
aux (1*3*2) 1 => aux (1*3*2*1) (1-1)
=> {- aux 2nd equation -}
aux (1*3*2*1) 0
=> {- aux 1st equation -}
1*3*2*1 => ... => 6
```

# Sublistas: take y drop

- `take n xs` devuelve el prefijo de `xs` de longitud `n`, o bien la lista `xs` si `n` es mayor o igual que la longitud de `xs`:

Prelude `take :: Int -> [a] -> [a]`

- `drop n xs` devuelve el sufijo de `xs` después de los primeros `n` elementos, o bien la lista vacía `[]` si `n` es mayor o igual que la longitud de `xs`:

Prelude `drop :: Int -> [a] -> [a]`

```
Prelude> take 2 [10,7,3,5]
[10,7]
```

```
Prelude> drop 1 [10,7,3,5]
[7,3,5]
```

```
Prelude> take 8 [10,7,3,5]
[10,7,3,5]
```

```
Prelude> drop 8 [10,7,3,5]
[]
```

# Funciones de Orden Superior

- Una **Función de Orden Superior** es una función que toma otra función como argumento o devuelve una función como resultado

[http://en.wikipedia.org/wiki/Higher-order\\_function](http://en.wikipedia.org/wiki/Higher-order_function)

- Son claves en la programación funcional para definir **código reutilizable**
- La misma función de orden superior puede usarse para diferentes propósitos dependiendo de la función pasada como argumento

# Aplicando una función a los elementos de una lista

- map aplica la misma función a todos los elementos de una lista, devolviendo la lista de los resultados:

[http://en.wikipedia.org/wiki/Map\\_\(higher-order\\_function\)](http://en.wikipedia.org/wiki/Map_(higher-order_function))

map es una función de orden superior: ¡su primer argumento es una función!

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

```
square :: Integer -> Integer
square x = x * x
```

```
Prelude even :: (Integral a) => a -> Bool
Prelude even x = mod x 2 == 0
```

square :: Integer -> Integer

```
Main> map square [1,2,3]
[1,4,9]
```

even :: Integer -> Bool

```
Main> map even [1,2,3,4]
[False,True,False,True]
```

# Seleccionando elementos de una lista

- filter toma un **predicado** (condición) y una lista, y devuelve una lista con los elementos de la original que hacen que el predicado devuelva **True**:

filter es una **función de orden superior**: ¡su argumento es una función!

[http://en.wikipedia.org/wiki/Filter\\_\(higher-order\\_function\)](http://en.wikipedia.org/wiki/Filter_(higher-order_function))

```
Prelude filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
| p x      = x : filter p xs
| otherwise = filter p xs
```

```
Prelude even :: (Integral a) -> a -> Bool
even x = mod x 2 == 0
```

```
Char isDigit :: Char -> Bool
isDigit x = x >= '0' && x <= '9'
```

```
Main> filter even [1,2,0,3]
[2,0]
```

```
Main> filter isDigit "p0ep1e"
"01"
```

even :: Integer -> Bool

isDigit :: Char -> Bool

# Lambda funciones

- Podemos pasar como argumento una función sin ponerle nombre
- Estas son las **funciones anónimas** o  $\lambda$ -funciones



```
Main> map (\x -> x*x) [1,2,3]  
[1,4,9]
```

La función que toma x para devolver  $x^2$

```
Main> map (\x -> x+10) [1,2,3,4]  
[11,12,13,14]
```

La función que toma x para devolver  $x+10$

```
Main> filter (\x -> mod x 2 == 0) [1,2,3,4]  
[2,4]
```

La función de x que comprueba si x es par

```
Main> filter (\x -> x > 2) [10,2,3,1]  
[10,3]
```

La función de x que comprueba si x es mayor que 2

# Secciones

- Podemos aplicar **un único** argumento a un operador binario
- Como resultado, obtenemos una función del argumento suprimido
- Esto se llama **sección** de un operador

The diagram illustrates the concept of function sections through four examples in a hypothetical programming language environment:

- ( $2^*$ ) :: Integer  $\rightarrow$  Integer**  
( $2^*$ ) es la forma abreviada de ( $\lambda x \rightarrow 2^*x$ )
- Main> map ( $2^*$ ) [1,2,3]  
[2,4,6]**  
( $^3$ ) :: Integer  $\rightarrow$  Integer  
( $^3$ ) es la forma abreviada de ( $\lambda x \rightarrow x^3$ )
- Main> map ( $^3$ ) [1,2,3,4]  
[1,8,27,64]**  
(<3) :: Integer  $\rightarrow$  Bool  
(<3) es la forma abreviada de ( $\lambda x \rightarrow x < 3$ )
- Main> filter (<3) [10,2,3,1]  
[2,1]**

# Secuencias aritméticas

## ■ Secuencias aritméticas:

- La diferencia de cualesquiera dos elementos consecutivos es constante

```
Prelude> [1..10]
```

```
[1,2,3,4,5,6,7,8,9,10]
```

La diferencia es 1

```
Prelude> [2,4..10]
```

```
[2,4,6,8,10]
```

La diferencia es 2

```
Prelude> [10,9..1]
```

```
[10,9,8,7,6,5,4,3,2,1]
```

La diferencia es -1

```
Prelude> [1,3..]
```

```
[1,3,5,7,9,11,13,15...]
```

Lista infinita con los naturales impares

# Listas por comprensión

[http://en.wikipedia.org/wiki/List\\_comprehension](http://en.wikipedia.org/wiki/List_comprehension)

- Permite definir listas con una sintaxis sencilla

- Generadores:

[ *expresión* | *patrón* <- *lista* ]

Normalmente una  
variable

Se lee como ∈

```
Prelude> [ x^2 | x <- [1..4] ]  
[1,4,9,16]
```

La lista de  $x^2$  para  
 $x \in [1,2,3,4]$

```
Prelude> [ even x | x <- [1..4] ]  
[False,True,False,True]
```

La lista de los valores  
booleanos  
correspondientes a los  
test de paridad de  $x$   
para  $x \in [1,2,3,4]$

```
Prelude> [ (x,even x) | x <- [1..4] ]  
[(1,False),(2,True),(3,False),(4,True)]
```

Un par para cada  
 $x \in [1,2,3,4]$

# Listas por comprensión (II)

## ■ Evaluación paso a paso:

```
[ x^2 | x <- [1,2,3] ]
=>
1^2 : [ x^2 | x <- [2,3] ]
=>
1^2 : 2^2 : [ x^2 | x <- [3] ]
=>
1^2 : 2^2 : 3^2 : [ x^2 | x <- [] ]
=>
1^2 : 2^2 : 3^2 : []
=>
1 : 4 : 9 : []
```

# Listas por comprensión (III)

## ■ Guardas:

[ *expresión* | *patrón* <- *lista*, *guarda* ]



Guarda: debe ser expresión booleana

Solo se tiene en cuenta los números positivos

Prelude> [ x | x <- [-1,2,3,-4], x>0 ]  
[2,3]

Prelude> [ x | x <- [1,2,3,4], even x ]  
[2,4]

Prelude> [ x^2 | x <- [1,2,3,4], even x ]  
[4,16]

Solo se tienen en cuenta los pares

Solo los pares se elevan al cuadrado y se devuelven

Es una combinación de map y filter

# Listas por comprensión (IV)

## ■ Definiciones locales:

[ *expresión<sub>1</sub>* | *patrón<sub>1</sub>* <- *lista*, let *patrón<sub>2</sub>* = *expresión<sub>2</sub>* ]

```
Prelude> [ (x,y) | x <- [1,2,3], let y = 2*x ]
[(1,2), (2,4), (3,6)]
```

```
Prelude> [ (x,2*x) | x <- [1,2,3] ]
[(1,2), (2,4), (3,6)]
```

Let *y* se refiere  
al doble de *x*

Sin definición  
local

# Listas por comprensión (V)

## ■ Generadores múltiples:

[ *expresión* | *patrón<sub>1</sub>* <- *lista<sub>1</sub>*, *patrón<sub>2</sub>* <- *lista<sub>2</sub>*, ... ]

Siendo 1 el valor de x, y  
toma todos los posibles  
valores

```
Prelude> [(x,y) | x <- [1,2,3], y <- [10,20]]  
[(1,10), (1,20), (2,10), (2,20), (3,10), (3,20)]
```

Producto  
Cartesiano

```
Prelude> [(x,y) | x <- [1,2,3], y <- [10,20], even (x+y)]  
[(2,10), (2,20)]
```

# Plegando listas

- `foldr`: reduce todos los elementos de una lista a un solo valor usando una función binaria (de plegado) y un valor inicial
- `foldr` se refiere a `fold right`. Los elementos de la lista se *plegarán* desde la derecha:

`Prelude`

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

Función binaria para operar los elementos

Resultado final

Lista a plegar

Valor inicial

```
foldr f z [x1, x2, ... xn]
=>
x1 `f` foldr f z [x2, ... xn]
=>
...
=>
x1 `f` (x2 `f` ... (xn-1 `f` (xn `f` z)) ... )
```

# Plegando listas (II)

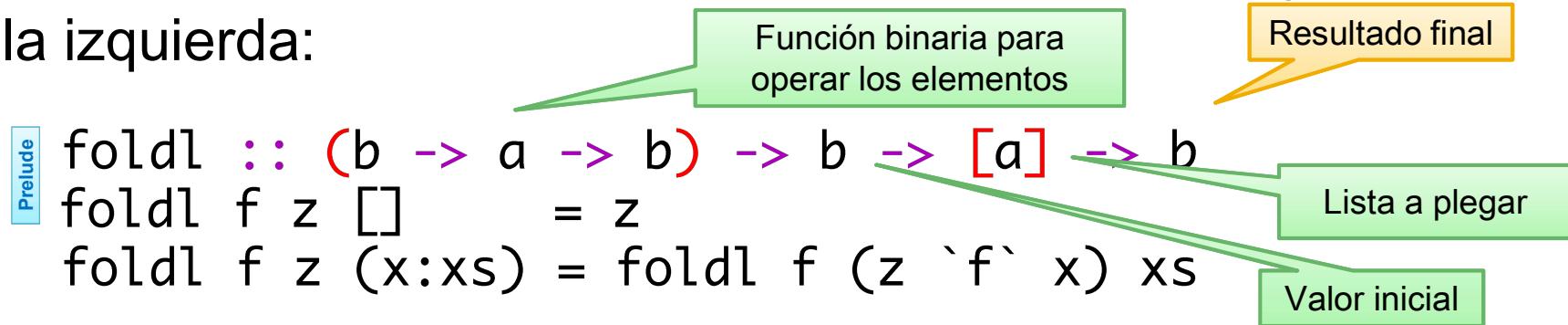
```
add :: (Num a) => a -> a -> a  
add x y = x + y
```



```
foldr add 0 [1, 2, 3, 4]  
=>  
...  
=>  
1 `add` ( 2 `add` ( 3 `add` ( 4 `add` 0 )))  
=>  
1 `add` ( 2 `add` ( 3 `add` 4 ))  
=>  
1 `add` ( 2 `add` 7 )  
=>  
1 `add` 9  
=>  
10
```

# Plegando listas (III)

- `foldl`: reduce todos los elementos de una lista a un solo valor usando una función binaria (de plegado) y un valor inicial  
[http://en.wikipedia.org/wiki/Fold\\_\(higher-order\\_function\)](http://en.wikipedia.org/wiki/Fold_(higher-order_function))
- `foldl` se refiere a **fold left**. Los elementos de la lista se *plegarán desde la izquierda*:



```
foldl f z [x1, x2, ... xn]
=>
foldl f (z `f` x1) [x2, ... xn]
=>
...
=>
( ... ((z `f` x1) `f` x2) ... `f` xn-1) `f` xn
```

# Plegando listas (IV)

```
add :: (Num a) => a -> a -> a  
add x y = x + y
```

foldl add 0 [1, 2, 3, 4]

=>  
...  
=> ((( 0 `add` 1 ) `add` 2 ) `add` 3 ) `add` 4  
=>  
(( 1 `add` 2 ) `add` 3 ) `add` 4  
=>  
( 3 `add` 3 ) `add` 4  
=>  
6 `add` 4  
=>  
10



# Plegando listas (y V)

- Muchas funciones del **Prelude** están definidas usando plegados:

Prelude sum :: (Num a) => [a] -> a  
sum xs = foldl (+) 0 xs

Prelude product :: (Num a) => [a] -> a  
product xs = foldl (\*) 1 xs

Prelude and :: [Bool] -> Bool  
and xs = foldr (&&) True xs

Prelude or :: [Bool] -> Bool  
or xs = foldr (||) False xs

Prelude concat :: [[a]] -> [a]  
concat XSS = foldr (++) [] XSS

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

```
Main> sum [1,2,3,4]
10
Main> product [1,2,3,4]
24
Main> and [1<3, even 2, 2 == 1+1]
True
Main> or [1<3, even 5, 6 == 1+1]
True
Main> concat [[1,2,3], [4], [5,6]]
[1,2,3,4,5,6]
Main> [1,2,3] ++ [5,6]
[1,2,3,5,6]
```

foldl (+) 0 xs

Para pasar un operador como parámetro a una función de orden superior hay que usar **parentesis**

# Parcialización

- La idea original es de:



<http://en.wikipedia.org/wiki/Currying>

Moses Schönfinkel y Haskell Curry

- Las funciones de varios argumentos pueden ser representadas usando funciones de un solo argumento
- Una función que toma  $n$  argumentos es representada por una función de orden superior que toma un solo argumento y devuelve una función que toma  $n-1$  argumentos
- Iterando este principio, cualquier función que toma  $n$  argumentos se representará por una cadena de  $n$  funciones, cada una de las cuales toma un solo argumento

# Parcialización (II)

- Consideremos la siguiente función:

$$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
$$f x y z = x + 2*y + 3*z$$

f toma tres enteros (x,y,z) devolviendo el entero  $(x + 2*y + 3*z)$



- La definición equivale a:

$$f :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int}))$$
$$f = \lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow x + 2*y + 3*z))$$

-> es asociativo a la derecha

f toma un entero (x) devolviendo la función que toma dos enteros (y,z) devolviendo el entero  $(x + 2*y + 3*z)$

- Que también es:

$$f :: \text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow (\text{Int} \rightarrow \text{Int})))$$
$$f = \lambda x \rightarrow (\lambda y \rightarrow (\lambda z \rightarrow (\lambda w \rightarrow x + 2*y + 3*z)))$$

f toma un entero (x) devolviendo la función que toma un entero (y) devolviendo la función que toma un entero (z) devolviendo el entero  $(x + 2*y + 3*z)$

# Parcialización (III)

- Las funciones parcializadas pueden usarse aplicando sus argumentos uno por uno
- Después de cada aplicación, se obtiene una función con el resto de argumentos

$$f :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$$
$$f\ x\ y\ z = x + 2*y + 3*z$$

- La expresión  $f\ 10$  tiene tipo  $\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$   
y se refiere a  $\lambda y\ z \rightarrow 10 + 2*y + 3*z$   
Una función que toma dos argumentos
- La expresión  $f\ 10\ 20$  tiene tipo  $\text{Int} \rightarrow \text{Int}$   
y se refiere a  $\lambda z \rightarrow 10 + 2*20 + 3*z$   
Una función que toma un argumento
- La expresión  $f\ 10\ 20\ 30$  tiene tipo  $\text{Int}$   
y se refiere a  $10 + 2*20 + 3*30$   
Un valor (función que toma cero argumentos)



# Tipificación en Haskell

- Regla fundamental de tipificación en Haskell:

- Si

$$f :: \alpha \rightarrow \beta$$

- y además

$$x :: \alpha$$

- entonces

$$f\ x :: \beta$$

Recuerde que :

even :: Int  $\rightarrow$  Bool  
10 :: Int

Por tanto:

even 10 :: Bool

# Tipificación en Haskell (II)

Regla fundamental de tipificación en Haskell:

- *Si*

$$f :: \alpha \rightarrow \beta \rightarrow \gamma$$

$\rightarrow$  es asociativo a la derecha

- *y además*

$$x :: \alpha$$

- *entonces*

$$f x :: \beta \rightarrow \gamma$$

- *Si además*

$$y :: \beta$$

- *entonces*

$$f x y :: \gamma$$

Recuerde que:

$$\text{take} :: \text{Int} \rightarrow [\alpha] \rightarrow [\alpha]$$

$$2 :: \text{Int}$$

$$[\text{True}, \text{True}, \text{False}] :: [\text{Bool}]$$

Por tanto:

$$\text{take } 2 :: [\alpha] \rightarrow [\alpha]$$

$$\text{take } 2 [\text{True}, \text{True}, \text{False}] :: [\text{Bool}]$$

# Parcialización (y VI)

## ■ Más ejemplos:

```
isMultipleOf 3 :: Int -> Bool
```

```
Main> filter (isMultipleOf 3) [1..10]  
[3,6,9]
```

```
Main> filter (isMultipleOf 2) [1..10]  
[2,4,6,8,10]
```

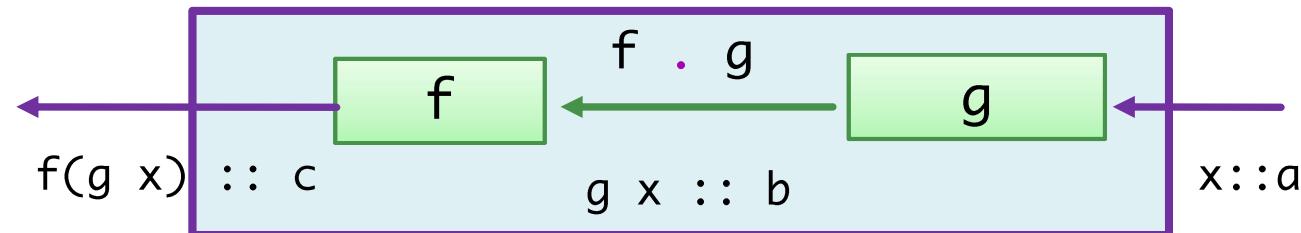
```
Main> map (+1) [2,4,6]  
[3,5,7]
```

```
map (+1) :: [Int] -> [Int]
```

```
Main> map (map (+1)) [[2,4,6], [5], [0,1]]  
[[3,5,7], [6], [1,2]]
```

# Composición de Funciones

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

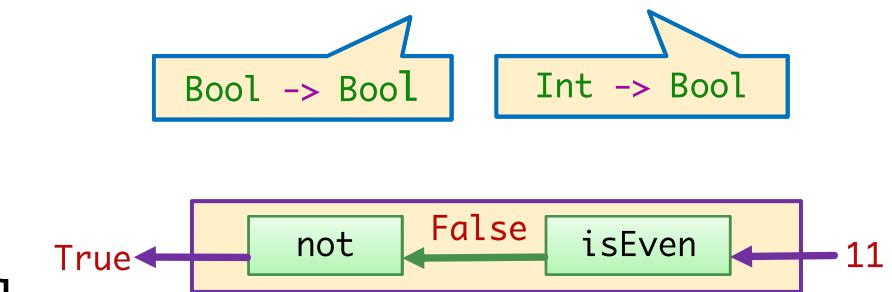


[http://en.wikipedia.org/wiki/Function\\_composition\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Function_composition_(computer_science))



- `(.)` es una función de orden superior que toma dos funciones y devuelve como resultado la función composición
- La ecuación puede ser reescrita como  $(.) f g x = f (g x)$
- donde vemos que la función compuesta `(.) f g` toma el argumento `x`, aplica `g` a éste, el resultado se pasa a la primera función `f` que devolverá el resultado final de la composición

`isOdd :: Int -> Bool  
isOdd = not . isEven`



# Tipos Algebraicos

[http://en.wikipedia.org/wiki/Algebraic\\_data\\_type](http://en.wikipedia.org/wiki/Algebraic_data_type)

- Podemos definir nuevos tipos de datos usando definiciones de Tipos Algebraicos (**data**)
- Cada definición proporciona uno o más **casos** para construir valores
- Cada caso tiene asociado una etiqueta (**Constructor de Datos**) y cero o más componentes

# Tipos Enumerados

- El nuevo tipo es un conjunto finito de valores

Constructores de Datos (valores): también comienzan con Mayúsculas

```
data Direction = North | South | East | West
```

Nuevo **Tipo**. Comienza con mayúscula

# Clases de Tipos

- Una Clase de Tipos es un conjunto abstracto de operaciones (funciones y operadores)
- Si un tipo de datos implementa estas operaciones podemos decir que es una instancia de la clase
- El concepto es similar a los interfaces de Java, pero puede incluir definiciones por defecto

# Clase Eq

- **Eq** es una clase de tipos que define los operadores de igualdad y desigualdad:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

$$\begin{aligned}x == y &= \text{not } (x /= y) \\x /= y &= \text{not } (x == y)\end{aligned}$$

Definiciones por defecto

En cada instancia de Eq hay que definir al menos uno de los operadores para romper la circularidad

# Igualdad para el tipo Direction

- Para definir la igualdad (y desigualdad) para el tipo Direction, debemos declarar la instancia:

```
data Direction = North | South | East | West
```

```
instance Eq Direction where
```

North	<code>==</code>	North	<code>= True</code>
South	<code>==</code>	South	<code>= True</code>
East	<code>==</code>	East	<code>= True</code>
West	<code>==</code>	West	<code>= True</code>
-	<code>==</code>	-	<code>= False</code>

```
Main> North == North  
True  
Main> North == South  
False  
Main> North /= South  
True
```

Basta definir `(==)` en la instancia. Para reducir `x /= y` se usa la definición por defecto que aparece en la clase

# Clase Ord

- `Ord` es la clase de tipos que define los operadores de ordenación:

```
data Ordering = LT | EQ | GT
```

LT = less than  
EQ = equal than  
GT = greater than

```
class Eq a => Ord a where
  (<) :: a -> a -> Bool
  (≤) :: a -> a -> Bool
  (>) :: a -> a -> Bool
  (≥) :: a -> a -> Bool
  max :: a -> a -> a
  min :: a -> a -> a
  compare :: a -> a -> Ordering
```

Definiciones por defecto.  
Hay que definir (≤) o bien compare

$x < y = x \leq y \ \&\& \ \text{not } (x == y)$   
 $x > y = y < x$

... otras definiciones por defecto

# Orden para Direction

- Para definir el siguiente orden en Direction

North < South < East < West

debemos declarar la instancia:

```
instance Ord Direction where
    North <= _      = True
    South <= North = False
    South <= _      = True
    East  <= North = False
    East  <= South = False
    East  <= _      = True
    West  <= North = False
    West  <= South = False
    West  <= East  = False
    West  <= _      = True
```

```
Main> North <= North
True
Main> West > South
True
Main> max West South
West
Main> compare West South
GT
```

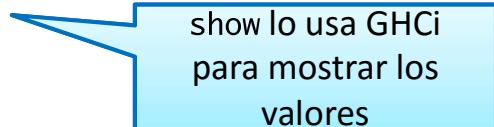
Solo definimos ( $\leq$ ) en la instancia.  
Para el resto de operadores usaremos  
las definiciones por defecto

# Clase Show

- **Show** es la clase de tipos que define cómo Haskell debe mostrar los datos:

```
class Show a where  
    show :: a -> String
```

```
instance Show Direction where  
    show North = "North"  
    show South = "South"  
    show East = "East"  
    show West = "West"
```



show lo usa GHCi  
para mostrar los  
valores

# Generación automática de instancias

- La instancias pueden a veces ser generadas automáticamente. Para ello debemos incluir la cláusula **deriving** en la definición **data** :

```
data Direction = North | South | East | West deriving (Show, Eq, Ord)
```

```
Main> show North  
"North"
```

```
Main> North == North  
True
```

```
Main> North == South  
False
```

```
Main> North < South  
True
```

```
Main> West > South  
True
```

Solo dos valores idénticos son considerados iguales

Los valores se ordenan según aparecen en la definición del tipo.  
Menor el de más a la izquierda

Las instancias para Show, Eq y Ord se generan automáticamente para este tipo de datos

# Tipos Unión

- Disponen de varios Constructores de Datos con un argumento:

```
data Degrees = Celsius Double | Fahrenheit Double deriving Show
```

Primer  
Constructor

Segundo  
Constructor

```
frozen :: Degrees -> Bool
frozen (Celsius c) = c <= 0
frozen (Fahrenheit f) = f <= 32
```

```
Main> toCelsius (Fahrenheit 32)
Celsius 0.0
```

```
Main> frozen (Celsius 20)
False
```

```
Main> :t Celsius
Celsius :: Double -> Degrees
```

Los Constructores de Datos  
pueden usarse en los Patrones ...

... y como  
funciones

```
toCelsius :: Degrees -> Degrees
toCelsius (Celsius c) = Celsius c
toCelsius (Fahrenheit f) = Celsius ((f-32) / 1.8)
```

```
toFahrenheit :: Degrees -> Degrees
toFahrenheit (Celsius c) = Fahrenheit (c*1.8 + 32)
toFahrenheit (Fahrenheit f) = Fahrenheit f
```



# Tipos Unión (II)

- Tipo unión polimórfico predefinido:

```
Prelude> data Either a b = Left a | Right b
```

```
list :: [Either Int Bool]
```

```
list = [ Left 1, Right True, Left 3, Left 4 ]
```

- Tipo opcional predefinido:

```
Prelude> data Maybe a = Nothing | Just a
```

```
Prelude> lookup 2 [(1,"hi"),(2,"good"),(3,"bye")]
Just "good"
```

```
Prelude> lookup 6 [(1,"hi"),(2,"good"),(3,"bye")]
Nothing
```

```
Prelude> lookup :: (Eq a) => a -> [(a, b)] -> Maybe b
```

```
lookup k' [] = Nothing
```

```
lookup k' ((k,v):xs)
```

```
| k' == k = Just v
```

```
| otherwise = lookup k' xs
```

Devuelve un valor x de tipo b (con **Just** x) o puede no devolver *nada* con **Nothing**

# Tipos Productos

- Disponen de un único constructor de datos con varios argumentos:

```
type Name = String
```

Sinónimos de tipo

```
type Surname = String
```

```
type Age = Int
```

Constructor

```
data Person = Pers Name Surname Age deriving Show
```

```
john :: Person
```

3 componentes

```
john = Pers "John" "Smith" 35
```

```
name :: Person -> Name
```

```
name (Pers nm _) = nm
```

el patrón nm es de tipo Name

```
surname :: Person -> Surname
```

```
surname (Pers _ snm _) = snm
```

snm :: Surname

```
age :: Person -> Age
```

```
age (Pers _ _ ag) = ag
```

ag :: Age

```
Main> name john  
"John"
```

```
Main> age john  
35
```

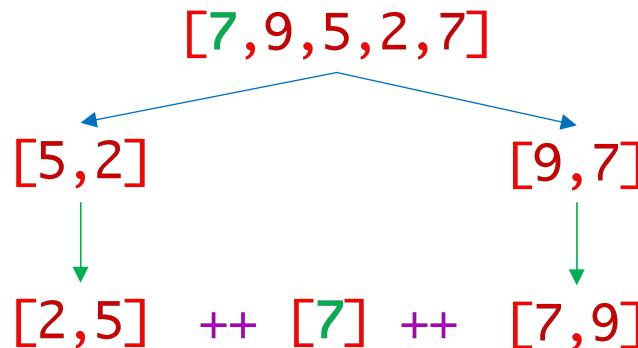
```
Main> surname (Pers "Mick" "Jagger" 70)  
"Jagger"
```

# Material Complementario

Para profundizar

# Ordenando una lista

- **Quick Sort**: el algoritmo de ordenación de C.A.R. Hoare (Premio Turing 1980)



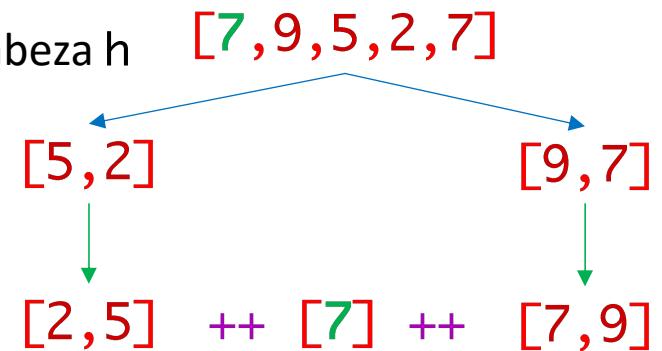
<http://en.wikipedia.org/wiki/Quicksort>

- Para ordenar una lista:
  - Tomamos un elemento de la lista, por ejemplo, la cabeza  $h$
  - Partimos la lista restante en dos sublistas: una con los menores que  $h$ , y otra con los mayores o iguales que  $h$
  - Recursivamente, ordenamos estas sublistas
  - Concatenamos la primera sublista ordenada con la cabeza y con la segunda sublista ordenada

# Ordenando una lista (II)

## Quick Sort:

- Tomamos un elemento de la lista, por ejemplo, la cabeza  $h$
- Partimos la lista restante en dos sublistas: una con los menores que  $h$ , y otra con los restantes
- Recursivamente, ordenamos estas sublistas
- Concatenamos la primera sublista ordenada con la cabeza y con la segunda sublista ordenada



```
qSort :: (Ord a) => [a] -> [a]
qSort []      = []
qSort (h:xs) = qSort ys ++ [h] ++ qSort zs
where
  ys = [ x | x <- xs, x < h ]
  zs = [ x | x <- xs, x >= h ]
```

```
Main> qSort [3,1,2,3]
[1,2,3,3]
Prelude> qSort "haskell"
"aehklls"
```

Anализar cómo trabaja

# Pruebas con QuickCheck (cont. IV)

```
import List
import Test.QuickCheck

isPermutationOf :: (Eq a) => [a] -> [a] -> Bool
xs `isPermutationOf` ys = null (xs \\\ ys) && null (ys \\\ xs)

sorted :: (Ord a) => [a] -> Bool
sorted [] = True
sorted [_] = True
sorted (x:xs@(y:_)) = (x<=y) && sorted xs
```

```
Main> [3,1,2,1] \\\ [1,3]
[2,1]
```

Diferencia de listas:  
import List

```
Main> [3,1,2] `isPermutationOf` [1,2,3]
True
```

isPermutationOf : ¿los argumentos tienen los mismos elementos, posiblemente en distinto orden ?

La lista ys obtenida con qSort xs debe estar ordenada y tener los mismos elementos que la original

- Una propiedad para probar qSort:

```
p_qSort xs = True ==> sorted ys && ys `isPermutationOf` xs
where ys = qSort xs
```

QuickCheck prueba nuestra propiedad con 100 listas de enteros aleatorios

```
Main> quickCheck (p_qSort :: [Int] -> Property)
+++ OK, passed 100 tests.
```

QuickCheck prueba nuestra propiedad con 100 listas de doubles aleatorios

```
Main> quickCheck (p_qSort :: [Double] -> Property)
+++ OK, passed 100 tests.
```

# Parcialización (IV)

```
-- tests whether x is a multiple of y  
isMultipleOf :: Int -> Int -> Bool  
isMultipleOf y x = mod x y == 0
```

isEven :: Int -> Bool      ¿Cuál es el tipo de isEven?  
isEven = isMultipleOf 2

¿Dónde está el argumento?

Está aquí

Como isEven = isMultipleOf 2,  
isEven debe tener el mismo  
Tipo que isMultipleOf 2

¿Cuál es el tipo de isMultipleOf 2?

isMultipleOf 2 :: Int -> Bool

Por tanto:

isEven :: Int -> Bool

# Parcialización (V)

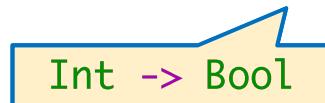
- Dos maneras diferentes de definir la misma función:

```
-- tests whether x is a multiple of y  
isMultipleOf :: Int -> Int -> Bool  
isMultipleOf y x = mod x y == 0
```

```
-- without currying  
isEven' :: Int -> Bool  
isEven' x = isMultipleOf 2 x
```



```
-- using currying  
isEven :: Int -> Bool  
isEven = isMultipleOf 2
```



```
isEven 10  
⇒ {- isEven definition -}  
isMultipleOf 2 10  
⇒ {- isMultipleOf definition -}  
mod 10 2 == 0  
⇒ {- mod definition -}  
0 == 0  
⇒  
True
```

# Inducción sobre listas

- Las pruebas no aseguran la corrección de los programas, y en ocasiones es necesario demostrar la corrección.
- Si una propiedad *prop* está definida sobre valores de un tipo inductivo *t*, podemos intentar demostrar la propiedad por inducción.
- En el caso de listas finitas podemos usar el esquema de **inducción estructural sobre listas**:

$$\forall xs . prop(xs) \iff \begin{cases} prop([]) & \text{Caso base} \\ \forall xs . prop(xs) \implies prop(x : xs) & \text{Hipótesis de inducción} \end{cases}$$

Paso inductivo



- **Caso base:** consiste en demostrar la propiedad para la lista vacía
- **Paso inductivo:** consiste en demostrar la propiedad para la lista  $(x:xs)$ . Para ello podemos asumir que la propiedad es cierta para la lista  $xs$  (**hipótesis de inducción**), o para cualquier sublista de  $(x:xs)$ .

# Un ejemplo de demostración por inducción sobre listas(I).

Es esencial describir con precisión la propiedad a demostrar

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

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

- Queremos probar que `map` no cambia la longitud de la lista:

$$\forall xs . \text{length}(\text{map } f \ xs) = \text{length } xs$$

- Usando inducción, debemos probar:

- Caso base:  $\text{length}(\text{map } f \ []) = \text{length } []$
- Paso inductivo:

Si  $\text{length}(\text{map } f \ xs) = \text{length } xs$

entonces  $\text{length}(\text{map } f \ (x:xs)) = \text{length} \ (x:xs)$

Hipótesis de inducción

# Un ejemplo de demostración por inducción sobre listas (II)

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

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

- Demostrando el caso base:

$$\text{length}(\text{map } f \text{ []}) = \text{length} \text{ []}$$

↔ {- 1<sup>a</sup> ecuación de map al miembro izquierdo -}

$$\text{length} \text{ []} = \text{length} \text{ []}$$

# Un ejemplo de demostración por inducción sobre listas (III)

- Demostrando el **paso inductivo**:

Si  $\text{length}(\text{map } f \text{ xs}) = \text{length xs}$   
entonces  $\text{length}(\text{map } f(x:xs)) = \text{length}(x:xs)$

Hipótesis de inducción

$\text{length} :: [\alpha] \rightarrow \text{Int}$   
 $\text{length } [] = 0$   
 $\text{length } (_:xs) = 1 + \text{length } xs$

$\text{map} :: (\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$   
 $\text{map } f [] = []$   
 $\text{map } f (x:xs) = f x : \text{map } f xs$

$$\text{length}(\text{map } f(x:xs)) = \text{length}(x:xs)$$

$\Leftrightarrow \{- \text{ 2ª ecuación de map (izda), y 2ª de length (derecha) -}\}$

$$\text{length}(f x : \text{map } f xs) = 1 + \text{length } xs$$

$\Leftrightarrow \{- \text{ 2ª de length (izda) -}\}$

$$1 + \text{length}(\text{map } f xs) = 1 + \text{length } xs$$

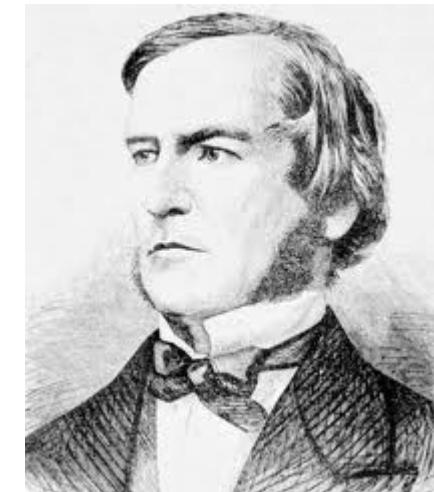
$\Leftrightarrow \{- \text{ Hipótesis de inducción (izda) y sustitutividad) -}\}$

$$1 + \text{length } xs = 1 + \text{length } xs$$

# Tipos Enumerados; los booleanos

Constructores de  
Datos

```
Prelude data Bool = False | True  
  
Prelude instance Eq Bool where  
    False == False = True  
    True == True = True  
    _ == _ = False  
  
Prelude instance Ord Bool where  
    False <= _ = True  
    True <= False = False  
    True <= _ = True  
  
Prelude instance Show Bool where  
    show False = "False"  
    show True = "True"
```



George Boole

[http://es.wikipedia.org/wiki/George\\_Boole](http://es.wikipedia.org/wiki/George_Boole)

# Booleanos (II)

```
Prelude not :: Bool -> Bool  
not False = True  
not True = False
```

```
Prelude infixr 3 &&  
(&&) :: Bool -> Bool -> Bool  
False && _ = False  
True && x = x
```

```
Prelude infixr 2 ||  
(||) :: Bool -> Bool -> Bool  
False || x = x  
True || _ = True
```

este operador es **estricto**  
solo en el primer argumento

Main> 1 < 2 || div 1 0 == 20  
True 😊

Los Constructores de  
Datos pueden ser  
usados como Patrones