

# Tema 7: Funciones de orden superior

## Informática (2014–15)

José A. Alonso Jiménez

Grupo de Lógica Computacional  
Departamento de Ciencias de la Computación e I.A.  
Universidad de Sevilla

## Tema 7: Funciones de orden superior

1. Funciones de orden superior
2. Procesamiento de listas
  - La función `map`
  - La función `filter`
3. Función de plegado por la derecha: `foldr`
4. Función de plegado por la izquierda: `foldl`
5. Composición de funciones
6. Caso de estudio: Codificación binaria y transmisión de cadenas
  - Cambio de bases
  - Codificación y decodificación

## Funciones de orden superior

- ▶ Una función es **de orden superior** si toma una función como argumento o devuelve una función como resultado.
- ▶ `(dosVeces f x)` es el resultado de aplicar `f` a `f x`. Por ejemplo,

<code>dosVeces (*3) 2</code>	<code>↔</code>	<code>18</code>
<code>dosVeces reverse [2,5,7]</code>	<code>↔</code>	<code>[2,5,7]</code>

---

```
dosVeces :: (a -> a) -> a -> a
dosVeces f x = f (f x)
```

---

- ▶ Prop: `dosVeces reverse = id`  
donde `id` es la función identidad.

---

```
Prelude
```

---

```
id :: a -> a
id x = x
```

---

## Funciones de orden superior

- ▶ Una función es **de orden superior** si toma una función como argumento o devuelve una función como resultado.
- ▶ `(dosVeces f x)` es el resultado de aplicar `f` a `f x`. Por ejemplo,

dosVeces (*3) 2	$\rightsquigarrow$	18
dosVeces reverse [2,5,7]	$\rightsquigarrow$	[2,5,7]

---

```
dosVeces :: (a -> a) -> a -> a
dosVeces f x = f (f x)
```

---

- ▶ Prop: `dosVeces reverse = id`  
donde `id` es la función identidad.

---

```
Prelude
```

---

```
id :: a -> a
id x = x
```

---

## Usos de las funciones de orden superior

- ▶ Definición de patrones de programación.
  - ▶ Aplicación de una función a todos los elementos de una lista.
  - ▶ Filtrado de listas por propiedades.
  - ▶ Patrones de recursión sobre listas.
- ▶ Diseño de lenguajes de dominio específico:
  - ▶ Lenguajes para procesamiento de mensajes.
  - ▶ Analizadores sintácticos.
  - ▶ Procedimientos de entrada/salida.
- ▶ Uso de las propiedades algebraicas de las funciones de orden superior para razonar sobre programas.

## Tema 7: Funciones de orden superior

1. Funciones de orden superior
2. Procesamiento de listas
  - La función `map`
  - La función `filter`
3. Función de plegado por la derecha: `foldr`
4. Función de plegado por la izquierda: `foldl`
5. Composición de funciones
6. Caso de estudio: Codificación binaria y transmisión de cadenas

## La función map: Definición

- ▶ `(map f xs)` es la lista obtenida aplicando `f` a cada elemento de `xs`. Por ejemplo,

```
map (*2) [3,4,7]      ~> [6,8,14]
map sqrt [1,2,4]     ~> [1.0,1.4142135623731,2.0]
map even [1..5]      ~> [False,True,False,True,False]
```

- ▶ Definición de map por comprensión:

---

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

---

- ▶ Definición de map por recursión:

---

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

---

## La función map: Definición

- ▶ `(map f xs)` es la lista obtenida aplicando `f` a cada elemento de `xs`. Por ejemplo,

```
map (*2) [3,4,7]      ~> [6,8,14]
map sqrt [1,2,4]     ~> [1.0,1.4142135623731,2.0]
map even [1..5]      ~> [False,True,False,True,False]
```

- ▶ Definición de map por comprensión:

---

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

---

- ▶ Definición de map por recursión:

---

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

---



## La función map: Definición

- ▶ `(map f xs)` es la lista obtenida aplicando `f` a cada elemento de `xs`. Por ejemplo,

```
map (*2) [3,4,7]      ~> [6,8,14]
map sqrt [1,2,4]     ~> [1.0,1.4142135623731,2.0]
map even [1..5]      ~> [False,True,False,True,False]
```

- ▶ Definición de map por comprensión:

---

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

---

- ▶ Definición de map por recursión:

---

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

---

## Relación entre sum y map

- ▶ La función sum:

---

```
Prelude
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

---

- ▶ Propiedad:  $\text{sum } (\text{map } (2*) \text{ xs}) = 2 * \text{sum } \text{xs}$
- ▶ Comprobación con QuickCheck:

---

```
prop_sum_map :: [Int] -> Bool
prop_sum_map xs = sum (map (2*) xs) == 2 * sum xs
```

---

```
*Main> quickCheck prop_sum_map
+++ OK, passed 100 tests.
```

## Relación entre sum y map

- ▶ La función sum:

---

```
Prelude  
sum :: [Int] -> Int  
sum []      = 0  
sum (x:xs) = x + sum xs
```

---

- ▶ Propiedad:  $\text{sum } (\text{map } (2*) \text{ xs}) = 2 * \text{sum } \text{xs}$
- ▶ Comprobación con QuickCheck:

---

```
prop_sum_map :: [Int] -> Bool  
prop_sum_map xs = sum (map (2*) xs) == 2 * sum xs
```

---

```
*Main> quickCheck prop_sum_map  
+++ OK, passed 100 tests.
```

## Tema 7: Funciones de orden superior

### 1. Funciones de orden superior

### 2. Procesamiento de listas

La función `map`

La función `filter`

### 3. Función de plegado por la derecha: `foldr`

### 4. Función de plegado por la izquierda: `foldl`

### 5. Composición de funciones

### 6. Caso de estudio: Codificación binaria y transmisión de cadenas

## La función `filter`

- ▶ `filter p xs` es la lista de los elementos de `xs` que cumplen la propiedad `p`. Por ejemplo,

```
filter even [1,3,5,4,2,6,1] ~ [4,2,6]
```

```
filter (>3) [1,3,5,4,2,6,1] ~ [5,4,6]
```

- ▶ Definición de `filter` por comprensión:

---

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

```
filter p xs = [x | x <- xs, p x]
```

---

- ▶ Definición de `filter` por recursión:

---

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

```
filter _ [] = []
```

```
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

---

## La función `filter`

- ▶ `filter p xs` es la lista de los elementos de `xs` que cumplen la propiedad `p`. Por ejemplo,

```
filter even [1,3,5,4,2,6,1] ~> [4,2,6]
```

```
filter (>3) [1,3,5,4,2,6,1] ~> [5,4,6]
```

- ▶ Definición de `filter` por comprensión:

---

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

```
filter p xs = [x | x <- xs, p x]
```

---

- ▶ Definición de `filter` por recursión:

---

```

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

```

---

## La función `filter`

- ▶ `filter p xs` es la lista de los elementos de `xs` que cumplen la propiedad `p`. Por ejemplo,

```
filter even [1,3,5,4,2,6,1] ~\~ [4,2,6]
```

```
filter (>3) [1,3,5,4,2,6,1] ~\~ [5,4,6]
```

- ▶ Definición de `filter` por comprensión:

---

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

```
filter p xs = [x | x <- xs, p x]
```

---

- ▶ Definición de `filter` por recursión:

---

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

```
filter _ [] = []
```

```
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

---

## Uso conjunto de `map` y `filter`

- ▶ `sumaCuadradosPares xs` es la suma de los cuadrados de los números pares de la lista `xs`. Por ejemplo,

```
| sumaCuadradosPares [1..5]  ~> 20
```

---

```
sumaCuadradosPares :: [Int] -> Int
sumaCuadradosPares xs = sum (map (^2) (filter even xs))
```

---

- ▶ Definición por comprensión:

---

```
sumaCuadradosPares' :: [Int] -> Int
sumaCuadradosPares' xs = sum [x^2 | x <- xs, even x]
```

---



## Uso conjunto de `map` y `filter`

- ▶ `sumaCuadradosPares xs` es la suma de los cuadrados de los números pares de la lista `xs`. Por ejemplo,

```
| sumaCuadradosPares [1..5]  ~> 20
```

---

```
sumaCuadradosPares :: [Int] -> Int
```

```
sumaCuadradosPares xs = sum (map (^2) (filter even xs))
```

---

- ▶ Definición por comprensión:

---

```
sumaCuadradosPares' :: [Int] -> Int
```

```
sumaCuadradosPares' xs = sum [x^2 | x <- xs, even x]
```

---

## Uso conjunto de `map` y `filter`

- ▶ `sumaCuadradosPares xs` es la suma de los cuadrados de los números pares de la lista `xs`. Por ejemplo,

```
| sumaCuadradosPares [1..5]  ~> 20
```

---

```
sumaCuadradosPares :: [Int] -> Int
```

```
sumaCuadradosPares xs = sum (map (^2) (filter even xs))
```

---

- ▶ Definición por comprensión:

---

```
sumaCuadradosPares' :: [Int] -> Int
```

```
sumaCuadradosPares' xs = sum [x^2 | x <- xs, even x]
```

---

## Predefinidas de orden superior para procesar listas

- ▶ `all p xs` se verifica si todos los elementos de `xs` cumplen la propiedad `p`. Por ejemplo,

```
| all odd [1,3,5]  ~> True
```

```
| all odd [1,3,6]  ~> False
```

- ▶ `any p xs` se verifica si algún elemento de `xs` cumple la propiedad `p`. Por ejemplo,

```
| any odd [1,3,5]  ~> True
```

```
| any odd [2,4,6]  ~> False
```

- ▶ `takeWhile p xs` es la lista de los elementos iniciales de `xs` que verifican el predicado `p`. Por ejemplo,

```
| takeWhile even [2,4,6,7,8,9] ~> [2,4,6]
```

- ▶ `dropWhile p xs` es la lista `xs` sin los elementos iniciales que verifican el predicado `p`. Por ejemplo,

```
| dropWhile even [2,4,6,7,8,9] ~> [7,8,9]
```

## Esquema básico de recursión sobre listas

- ▶ Ejemplos de definiciones recursivas:

---

```
Prelude
```

---

```
sum []           = 0
sum (x:xs)       = x + sum xs
product []       = 1
product (x:xs)   = x * product xs
or []            = False
or (x:xs)        = x || or xs
and []          = True
and (x:xs)       = x && and xs
```

---

- ▶ Esquema básico de recursión sobre listas:

---

```
f []           = v
f (x:xs)       = x 'op' (f xs)
```

---

## El patrón `foldr`

- Redefiniciones con el patrón `foldr`

---

```
sum      = foldr (+) 0
product = foldr (*) 1
or       = foldr (||) False
and      = foldr (&&) True
```

---

- Definición del patrón `foldr`

---

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

---

## El patrón `foldr`

- Redefiniciones con el patrón `foldr`

---

```
sum      = foldr (+) 0
product = foldr (*) 1
or       = foldr (||) False
and      = foldr (&&) True
```

---

- Definición del patrón `foldr`

---

```
_____ Prelude _____
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []      = v
foldr f v (x:xs) = f x (foldr f v xs)
```

---

## Visión no recursiva de `foldr`

- ▶ Cálculo con `sum`:

`sum [2,3,5]`

`= foldr (+) 0 [2,3,5]` [def. de `sum`]

`= foldr (+) 0 2:(3:(5:[]))` [notación de lista]

`= 2+(3+(5+0))` [sustituir `(:)` por `(+)` y `[]` por `0`]

`= 10` [aritmética]

- ▶ Cálculo con `sum`:

`product [2,3,5]`

`= foldr (*) 1 [2,3,5]` [def. de `sum`]

`= foldr (*) 1 2:(3:(5:[]))` [notación de lista]

`= 2*(3*(5*1))` [sustituir `(:)` por `(*)` y `[]` por `1`]

`= 30` [aritmética]

- ▶ Cálculo de `foldr f v xs`

Sustituir en `xs` los `(:)` por `f` y `[]` por `v`.

## Definición de la longitud mediante `foldr`

- ▶ Ejemplo de cálculo de la longitud:

$$\begin{aligned}
 & \text{longitud } [2,3,5] \\
 = & \text{longitud } 2:(3:(5:[])) \\
 = & \quad \quad \quad 1+(1+(1+0)) && \text{[Sustituciones]} \\
 = & 3
 \end{aligned}$$

- ▶ Sustituciones:
  - ▶ los `(:)` por `(\x y -> 1+y)`
  - ▶ la `[]` por `0`
- ▶ Definición de `length` usando `foldr`

---

```

longitud :: [a] -> Int
longitud = foldr (\x y -> 1+y) 0

```

---



## Definición de la longitud mediante `foldr`

- ▶ Ejemplo de cálculo de la longitud:

```
longitud [2,3,5]
= longitud 2:(3:(5:[]))
=          1+(1+(1+0))      [Sustituciones]
= 3
```

- ▶ Sustituciones:
  - ▶ los `(:)` por `(\x y -> 1+y)`
  - ▶ la `[]` por `0`
- ▶ Definición de `length` usando `foldr`

---

```
longitud :: [a] -> Int
longitud = foldr (\x y -> 1+y) 0
```

---

## Definición de la inversa mediante `foldr`

- ▶ Ejemplo de cálculo de la inversa:

```

  inversa [2,3,5]
= inversa 2:(3:(5:[]))
=          (([] ++ [5]) ++ [3]) ++ [2]  [Sustituciones]
= [5,3,2]

```

- ▶ Sustituciones:

- ▶ los `(:)` por `(\x y -> y ++ [x])`
- ▶ la `[]` por `[]`

- ▶ Definición de `inversa` usando `foldr`

---

```

inversa :: [a] -> [a]
inversa = foldr (\x y -> y ++ [x]) []

```

---

## Definición de la inversa mediante `foldr`

- ▶ Ejemplo de cálculo de la inversa:

```

| inversa [2,3,5]
| = inversa 2:(3:(5:[]))
| =          (([] ++ [5]) ++ [3]) ++ [2]  [Sustituciones]
| = [5,3,2]

```

- ▶ Sustituciones:

- ▶ los `(:)` por `(\x y -> y ++ [x])`
- ▶ la `[]` por `[]`

- ▶ Definición de `inversa` usando `foldr`

---

```
inversa :: [a] -> [a]
```

```
inversa = foldr (\x y -> y ++ [x]) []
```

---

## Definición de la concatenación mediante `foldr`

- ▶ Ejemplo de cálculo de la concatenación:

```

conc [2,3,5] [7,9]
= conc 2:(3:(5:[])) [7,9]
=      2:(3:(5:[7,9]))      [Sustituciones]
= [2,3,5,7,9]

```

- ▶ Sustituciones:
  - ▶ los `(:)` por `(:)`
  - ▶ la `[]` por `ys`
- ▶ Definición de `conc` usando `foldr`

---

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

---

## Definición de la concatenación mediante `foldr`

- ▶ Ejemplo de cálculo de la concatenación:

$$\begin{aligned}
 & \text{conc } [2,3,5] \ [7,9] \\
 = & \text{conc } 2:(3:(5:[])) \ [7,9] \\
 = & \quad 2:(3:(5:[7,9])) \quad \quad \quad \text{[Sustituciones]} \\
 = & [2,3,5,7,9]
 \end{aligned}$$

- ▶ Sustituciones:
  - ▶ los `(:)` por `(:)`
  - ▶ la `[]` por `ys`
- ▶ Definición de `conc` usando `foldr`

---

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

---

## Definición de suma de lista con acumuladores

- Definición de suma con acumuladores:

---

```
suma :: [Integer] -> Integer
suma = sumaAux 0
  where sumaAux v []      = v
        sumaAux v (x:xs) = sumaAux (v+x) xs
```

---

- Cálculo con suma:

```
suma [2,3,7] = sumaAux 0 [2,3,7]
             = sumaAux (0+2) [3,7]
             = sumaAux 2 [3,7]
             = sumaAux (2+3) [7]
             = sumaAux 5 [7]
             = sumaAux (5+7) []
             = sumaAux 12 []
             = 12
```

## Definición de suma de lista con acumuladores

- Definición de suma con acumuladores:

---

```
suma :: [Integer] -> Integer
suma = sumaAux 0
  where sumaAux v []      = v
        sumaAux v (x:xs) = sumaAux (v+x) xs
```

---

- Cálculo con suma:

```
suma [2,3,7] = sumaAux 0 [2,3,7]
             = sumaAux (0+2) [3,7]
             = sumaAux 2 [3,7]
             = sumaAux (2+3) [7]
             = sumaAux 5 [7]
             = sumaAux (5+7) []
             = sumaAux 12 []
             = 12
```

## Definición de suma de lista con acumuladores

- Definición de suma con acumuladores:

---

```
suma :: [Integer] -> Integer
suma = sumaAux 0
  where sumaAux v []      = v
        sumaAux v (x:xs) = sumaAux (v+x) xs
```

---

- Cálculo con suma:

```
suma [2,3,7] = sumaAux 0 [2,3,7]
             = sumaAux (0+2) [3,7]
             = sumaAux 2 [3,7]
             = sumaAux (2+3) [7]
             = sumaAux 5 [7]
             = sumaAux (5+7) []
             = sumaAux 12 []
             = 12
```



## Patrón de definición de recursión con acumulador

- ▶ Patrón de definición (generalización de `sumaAux`):

---

```
f v []      = v
f v (x:xs) = f (v*x) xs
```

---

- ▶ Definición con el patrón `foldl`:

---

```
suma      = foldl (+) 0
product  = foldl (*) 1
or        = foldl (||) False
and       = foldl (&&) True
```

---

## Definición de `foldl`

- Definición de `foldl`:

---

Prelude

---

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

---

- Diferencia entre `foldr` y `foldl`:

$$\begin{aligned} (\text{foldr } (-) \ 0) \ [3,4,2] &= 3 - (4 - (2 - 0)) = 1 \\ (\text{foldl } (-) \ 0) \ [3,4,2] &= ((0 - 3) - 4) - 2 = -9 \end{aligned}$$

## Definición de `foldl`

- Definición de `foldl`:

---

Prelude

---

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

---

- Diferencia entre `foldr` y `foldl`:

$$\begin{aligned} (\text{foldr } (-) 0) [3,4,2] &= 3 - (4 - (2 - 0)) = 1 \\ (\text{foldl } (-) 0) [3,4,2] &= ((0 - 3) - 4) - 2 = -9 \end{aligned}$$

## Definición de foldl

- Definición de foldl:

---

Prelude

---

```

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

```

---

- Diferencia entre foldr y foldl:

$$\begin{array}{l}
 | \text{(foldr } (-) \text{ 0) [3,4,2] = } 3 - (4 - (2 - 0)) = 1 \\
 | \text{(foldl } (-) \text{ 0) [3,4,2] = } ((0 - 3) - 4) - 2 = -9
 \end{array}$$

## Composición de funciones

- Definición de la composición de dos funciones:

---

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

---

- Uso de composición para simplificar definiciones:
  - Definiciones sin composición:

---

```
par n                = not (impar n)
doVeces f x          = f (f x )
sumaCuadradosPares ns = sum (map (^2) (filter even ns))
```

---

- Definiciones con composición:

---

```
par                = not . impar
dosVeces f         = f . f
sumaCuadradosPares = sum . map (^2) . filter even
```

## Composición de funciones

- ▶ Definición de la composición de dos funciones:

---

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

---

- ▶ Uso de composición para simplificar definiciones:
  - ▶ Definiciones sin composición:

---

```
par n                = not (impar n)
doVeces f x         = f (f x )
sumaCuadradosPares ns = sum (map (^2) (filter even ns))
```

---

- ▶ Definiciones con composición:

---

```
par                = not . impar
dosVeces f         = f . f
sumaCuadradosPares = sum . map (^2) . filter even
```

## Composición de funciones

- ▶ Definición de la composición de dos funciones:

---

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

---

- ▶ Uso de composición para simplificar definiciones:
  - ▶ Definiciones sin composición:

---

```
par n                = not (impar n)
doVeces f x          = f (f x )
sumaCuadradosPares ns = sum (map (^2) (filter even ns))
```

---

- ▶ Definiciones con composición:

---

```
par                = not . impar
dosVeces f         = f . f
sumaCuadradosPares = sum . map (^2) . filter even
```

## Composición de una lista de funciones

- ▶ La función identidad:

---

```
Prelude
id :: a -> a
id = \x -> x
```

---

- ▶ `(composicionLista fs)` es la composición de la lista de funciones `fs`. Por ejemplo,

```
composicionLista [(*) , (^2)] 3    ~> 18
composicionLista [ (^2) , (*) ] 3  ~> 36
composicionLista [(/9) , (^2) , (*)] 3 ~> 4.0
```

---

```
composicionLista :: [a -> a] -> (a -> a)
composicionLista = foldr (.) id
```

---



## Composición de una lista de funciones

- ▶ La función identidad:

---

```
Prelude
id :: a -> a
id = \x -> x
```

---

- ▶ (`composicionLista fs`) es la composición de la lista de funciones `fs`. Por ejemplo,

```
composicionLista [(*) , (^2)] 3    ~> 18
composicionLista [(^2) , (*)] 3    ~> 36
composicionLista [(/9) , (^2) , (*)] 3 ~> 4.0
```

---

```
composicionLista :: [a -> a] -> (a -> a)
composicionLista = foldr (.) id
```

---

## Caso de estudio: Codificación binaria y transmisión de cadenas

- ▶ Objetivos:
  1. Definir una función que convierta una cadena en una lista de ceros y unos junto con otra función que realice la conversión opuesta.
  2. Simular la transmisión de cadenas mediante ceros y unos.
- ▶ Los números binarios se representan mediante listas de bits en orden inverso. Un bit es cero o uno. Por ejemplo, el número 1101 se representa por [1,0,1,1].
- ▶ El tipo `Bit` es el de los bits.

---

```
type Bit = Int
```

---

## Tema 7: Funciones de orden superior

1. Funciones de orden superior
2. Procesamiento de listas
3. Función de plegado por la derecha: `foldr`
4. Función de plegado por la izquierda: `foldl`
5. Composición de funciones
6. Caso de estudio: Codificación binaria y transmisión de cadenas
  - Cambio de bases
  - Codificación y decodificación

## Cambio de bases: De binario a decimal

- `(bin2int x)` es el número decimal correspondiente al número binario `x`. Por ejemplo,

```
| bin2int [1,0,1,1]  ~>  13
```

El cálculo es

```
|   bin2int [1,0,1,1]
= bin2int 1:(0:(1:(1:[])))
=           1+2*(0+2*(1+2*(1+2*0)))
= 13
```

---

```
bin2int :: [Bit] -> Int
```

```
bin2int = foldr (\x y -> x + 2*y) 0
```

---

## Cambio de bases: De binario a decimal

- `(bin2int x)` es el número decimal correspondiente al número binario `x`. Por ejemplo,

```
| bin2int [1,0,1,1]  ~>  13
```

El cálculo es

```
|   bin2int [1,0,1,1]
=  bin2int 1:(0:(1:(1:[])))
=           1+2*(0+2*(1+2*(1+2*0)))
=  13
```

---

```
bin2int :: [Bit] -> Int
```

```
bin2int = foldr (\x y -> x + 2*y) 0
```

---

## Cambio de base: De decimal a binario

- ▶ `(int2bin x)` es el número binario correspondiente al número decimal `x`. Por ejemplo,

```
| int2bin 13  ~>  [1,0,1,1]
```

---

```
int2bin :: Int -> [Bit]
int2bin n | n < 2      = [n]
          | otherwise = n `mod` 2 : int2bin (n `div` 2)
```

---

Por ejemplo,

```
| int2bin 13
= 13 `mod` 2 : int2bin (13 `div` 2)
= 1 : int2bin (6 `div` 2)
= 1 : (6 `mod` 2 : int2bin (6 `div` 2))
= 1 : (0 : int2bin 3)
= 1 : (0 : (3 `mod` 2 : int2bin (3 `div` 2)))
= 1 : (0 : (1 : int2bin 1))
= 1 : (0 : (1 : (1 : int2bin 0)))
= 1 : (0 : (1 : (1 : [])))
= [1,0,1,1]
```

## Cambio de base: De decimal a binario

- ▶ `(int2bin x)` es el número binario correspondiente al número decimal `x`. Por ejemplo,

```
| int2bin 13  ~>  [1,0,1,1]
```

---

```
int2bin :: Int -> [Bit]
```

```
int2bin n | n < 2      = [n]
```

```
          | otherwise = n `mod` 2 : int2bin (n `div` 2)
```

---

Por ejemplo,

```
| int2bin 13
= 13 `mod` 2 : int2bin (13 `div` 2)
= 1 : int2bin (6 `div` 2)
= 1 : (6 `mod` 2 : int2bin (6 `div` 2))
= 1 : (0 : int2bin 3)
= 1 : (0 : (3 `mod` 2 : int2bin (3 `div` 2)))
= 1 : (0 : (1 : int2bin 1))
= 1 : (0 : (1 : (1 : int2bin 0)))
= 1 : (0 : (1 : (1 : [])))
= [1,0,1,1]
```

## Cambio de base: Comprobación de propiedades

- ▶ Propiedad: Al pasar un número natural a binario con `int2bin` y el resultado a decimal con `bin2int` se obtiene el número inicial.

---

```
prop_int_bin :: Int -> Bool
prop_int_bin x =
    bin2int (int2bin y) == y
  where y = abs x
```

---

- ▶ Comprobación:

```
*Main> quickCheck prop_int_bin
+++ OK, passed 100 tests.
```



## Cambio de base: Comprobación de propiedades

- ▶ Propiedad: Al pasar un número natural a binario con `int2bin` y el resultado a decimal con `bin2int` se obtiene el número inicial.

---

```
prop_int_bin :: Int -> Bool
prop_int_bin x =
    bin2int (int2bin y) == y
  where y = abs x
```

---

- ▶ Comprobación:

```
*Main> quickCheck prop_int_bin
+++ OK, passed 100 tests.
```

## Tema 7: Funciones de orden superior

1. Funciones de orden superior
2. Procesamiento de listas
3. Función de plegado por la derecha: `foldr`
4. Función de plegado por la izquierda: `foldl`
5. Composición de funciones
6. Caso de estudio: Codificación binaria y transmisión de cadenas
  - Cambio de bases
  - Codificación y descodificación

## Creación de octetos

- ▶ Un octeto es un grupo de ocho bits.
- ▶ (`creaOcteto bs`) es el octeto correspondiente a la lista de bits `bs`; es decir, los 8 primeros elementos de `bs` si su longitud es mayor o igual que 8 y la lista de 8 elementos añadiendo ceros al final de `bs` en caso contrario. Por ejemplo,

```
Main*> creaOcteto [1,0,1,1,0,0,1,1,1,0,0,0]
[1,0,1,1,0,0,1,1]
Main*> creaOcteto [1,0,1,1]
[1,0,1,1,0,0,0,0]
```

---

```
creaOcteto :: [Bit] -> [Bit]
```

```
creaOcteto bs = take 8 (bs ++ repeat 0)
```

---

donde (`repeat x`) es una lista infinita cuyo único elemento es `x`.

## Codificación

- ▶ `(codifica c)` es la codificación de la cadena `c` como una lista de bits obtenida convirtiendo cada carácter en un número Unicode, convirtiendo cada uno de dichos números en un octeto y concatenando los octetos para obtener una lista de bits. Por ejemplo,

```
*Main> codifica "abc"  
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

---

```
codifica :: String -> [Bit]  
codifica = concat . map (creaOcteto . int2bin . ord)
```

---

donde `(concat xss)` es la lista obtenida concatenando la lista de listas `xss`.

## Codificación

- ▶ `(codifica c)` es la codificación de la cadena `c` como una lista de bits obtenida convirtiendo cada carácter en un número Unicode, convirtiendo cada uno de dichos números en un octeto y concatenando los octetos para obtener una lista de bits. Por ejemplo,

```
*Main> codifica "abc"  
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

---

```
codifica :: String -> [Bit]  
codifica = concat . map (creaOcteto . int2bin . ord)
```

---

donde `(concat xss)` es la lista obtenida concatenando la lista de listas `xss`.

## Codificación

► Ejemplo de codificación,

```

codifica "abc"
= concat . map (creaOcteto . int2bin . ord) "abc"
= concat . map (creaOcteto . int2bin . ord) ['a','b','c']
= concat [creaOcteto . int2bin . ord 'a',
          creaOcteto . int2bin . ord 'b',
          creaOcteto . int2bin . ord 'c']
= concat [creaOcteto [1,0,0,0,0,1,1],
          creaOcteto [0,1,0,0,0,1,1],
          creaOcteto [1,1,0,0,0,1,1]]
= concat [[1,0,0,0,0,1,1,0],
          [0,1,0,0,0,1,1,0],
          [1,1,0,0,0,1,1,0]]
= [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]

```

## Separación de octetos

- ▶ `(separaOctetos bs)` es la lista obtenida separando la lista de bits `bs` en listas de 8 elementos. Por ejemplo,

```
*Main> separaOctetos [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0]
[[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0]]
```

---

```
separaOctetos :: [Bit] -> [[Bit]]
separaOctetos [] = []
separaOctetos bs =
    take 8 bs : separaOctetos (drop 8 bs)
```

---

## Separación de octetos

- ▶ `(separaOctetos bs)` es la lista obtenida separando la lista de bits `bs` en listas de 8 elementos. Por ejemplo,

```
*Main> separaOctetos [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0]
[[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0]]
```

---

```
separaOctetos :: [Bit] -> [[Bit]]
```

```
separaOctetos [] = []
```

```
separaOctetos bs =
```

```
    take 8 bs : separaOctetos (drop 8 bs)
```

---



## Descodificación

- `(descodifica bs)` es la cadena correspondiente a la lista de bits

bs. Por ejemplo,

```
*Main> descodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
"abc"
```

---

```
descodifica :: [Bit] -> String
descodifica = map (chr . bin2int) . separaOctetos
```

---

Por ejemplo,

```
descodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
= (map (chr . bin2int) . separaOctetos)
  [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
= map (chr . bin2int) [[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0],[1,1,0,0,0,1,1,0]]
= [(chr . bin2int) [1,0,0,0,0,1,1,0],
   (chr . bin2int) [0,1,0,0,0,1,1,0],
   (chr . bin2int) [1,1,0,0,0,1,1,0]]
= [chr 97, chr 98, chr 99]
= "abc"
```

## Descodificación

- `(descodifica bs)` es la cadena correspondiente a la lista de bits

bs. Por ejemplo,

```
*Main> descodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
"abc"
```

---

```
descodifica :: [Bit] -> String
```

```
descodifica = map (chr . bin2int) . separaOctetos
```

---

Por ejemplo,

```
descodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
= (map (chr . bin2int) . separaOctetos)
  [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
= map (chr . bin2int) [[1,0,0,0,0,1,1,0], [0,1,0,0,0,1,1,0], [1,1,0,0,0,1,1,0]]
= [(chr . bin2int) [1,0,0,0,0,1,1,0],
   (chr . bin2int) [0,1,0,0,0,1,1,0],
   (chr . bin2int) [1,1,0,0,0,1,1,0]]
= [chr 97, chr 98, chr 99]
= "abc"
```

## Transmisión

- ▶ Los canales de transmisión pueden representarse mediante funciones que transforman cadenas de bits en cadenas de bits.
- ▶ `(transmite c t)` es la cadena obtenida transmitiendo la cadena `t` a través del canal `c`. Por ejemplo,

```
*Main> transmite id "Texto por canal correcto"  
"Texto por canal correcto"
```

---

```
transmite :: ([Bit] -> [Bit]) -> String -> String  
transmite canal = descodifica . canal . codifica
```

---

## Corrección de la transmisión

- ▶ Propiedad: Al transmitir cualquier cadena por el canal identidad se obtiene la cadena.

---

```
prop_transmite :: String -> Bool
prop_transmite cs =
    transmite id cs == cs
```

---

- ▶ Comprobación de la corrección:

```
*Main> quickCheck prop_transmite
+++ OK, passed 100 tests.
```

## Corrección de la transmisión

- ▶ Propiedad: Al transmitir cualquier cadena por el canal identidad se obtiene la cadena.

---

```
prop_transmite :: String -> Bool
prop_transmite cs =
  transmite id cs == cs
```

---

- ▶ Comprobación de la corrección:

```
*Main> quickCheck prop_transmite
+++ OK, passed 100 tests.
```

## Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
  - ▶ Cap. 4: Listas.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
  - ▶ Cap. 7: Higher-order functions.
3. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
  - ▶ Cap. 8: Funciones de orden superior y polimorfismo.
4. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
  - ▶ Cap. 9: Generalization: patterns of computation.
  - ▶ Cap. 10: Functions as values.