Programación con listas

El tipo lista

- La lista representa colecciones de objetos homogéneas (todos los objetos han de tener el mismo tipo).

$$\textbf{infixr}\ 5\ :$$
$$\textbf{data}\ [a] = [\,] \mid a : [a] \quad -- \text{pseudocódigo}$$

$$[\,] :: [a]$$
$$(:) :: a \to [a] \to [a]$$

$$\text{PRELUDE}> True : [\,]$$
$$[True] :: [Bool]$$
$$\text{PRELUDE}> False : (True : [\,])$$
$$[False, True] :: [Bool]$$
$$\text{PRELUDE}> \text{'a'} : (True : [\,])$$

$$\begin{array}{ll} ERROR & : Type\ error\ in\ application \\ *** Expression & : \text{'a'} : True : [\,] \\ *** Term & : \text{'a'} \\ *** Type & : Char \\ *** Does\ not\ match & : Bool \end{array}$$

- $(:)$ asociativo a la derecha.

$$\text{PRELUDE}> 1 : (2 : (3 : [\,]))$$
$$[1, 2, 3] :: [Integer]$$
$$\text{PRELUDE}> 1 : 2 : 3 : [\,]$$
$$[1, 2, 3] :: [Integer]$$
$$\text{PRELUDE}> [1, 2, 3]$$
$$[1, 2, 3] :: [Integer]$$

Secuencias aritméticas. La clase *Enum*

- *Enum* es la clase de PRELUDE de los tipos *enumerables*.

```
class Enum a where

  succ, pred          :: a → a
  toEnum              :: Int → a
  fromEnum            :: a → Int
  enumFrom            :: a → [a]
  enumFromThen        :: a → a → [a]
  enumFromTo          :: a → a → [a]
  enumFromThenTo      :: a → a → a → [a]

  succ                    = toEnum . (1+) . fromEnum
  pred                    = toEnum . subtract 1 . fromEnum
  enumFrom x              = map toEnum [ fromEnum x .. ]
  enumFromTo x y          = map toEnum [ fromEnum x .. fromEnum y]
  enumFromThen x y        = map toEnum [ fromEnum x, fromEnum y ..]
  enumFromThenTo x y z    = map toEnum [ fromEnum x, fromEnum y .. fromEnum z ]
```

- Ejemplos

```
data Día  = Lunes | Martes | Miércoles | Jueves
          | Viernes | Sábado | Domingo      deriving (Show, Enum)
```

```
MAIN> succ Lunes
Martes :: Día
MAIN> pred Miércoles
Martes :: Día
```

```
MAIN> fromEnum Lunes
0 :: Int
MAIN> toEnum 4 :: Día
Viernes :: Día
```

- Creación de una instancia para Nat

```
data Nat = Cero | Suc Nat deriving Show

instance Enum Nat where

  toEnum 0       =Cero
  toEnum (n+1)   =Suc (toEnum n)

  fromEnum Cero    =0
  fromEnum (Suc n) =1 + fromEnum n


Main> succ Cero
Suc Cero :: Nat
Main> fromEnum (Suc Cero)
1 :: Int
Main> toEnum 5 :: Nat
Suc (Suc (Suc (Suc (Suc Cero)))) :: Nat
```

- o de otra forma (vía $iter$ en p.73)

```
instance Enum Nat where

  toEnum =iter (λ_ q → Suc q) Cero

  fromEnum =foldNat (+1) 0
```

- Secuencias aritméticas (para instancias de la clase $Enum$).

```
Prelude> [1 .. 10]
[1,2,3,4,5,6,7,8,9,10] :: [Integer]

Prelude> [1,3..11]
[1,3,5,7,9,11] :: [Integer]

Prelude> [10,9..1]
[10,9,8,7,6,5,4,3,2,1] :: [Integer]

Prelude> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15{Interrupted!}

Main> [Cero .. ]
[Cero,Suc Cero,Suc (Suc Cero),Suc (Suc (Suc Cero)),
Suc (Suc (Suc (Suc Cero)))){Interrupted!}

Main> [Jueves .. ]
[Jueves,Viernes,Sábado,Domingo] :: [Día]
```

- Las secuencias aritméticas son implementadas mediante los métodos: *enumFrom*, *enumFromThen*, *enumFromTo* y *enumFromThenTo* de la clase *Enum*.

Semántica de las secuencias aritméticas

$$
\begin{array}{ll}
[\ e_1\ ..\ ] & \rightsquigarrow\ enumFrom\ e_1 \\
[\ e_1\ ..\ e_2\ ] & \rightsquigarrow\ enumFromTo\ e_1\ e_2 \\
[\ e_1, e_2\ ..\ ] & \rightsquigarrow\ enumFromThen\ e_1\ e_2 \\
[\ e_1, e_2\ ..\ e_3\ ] & \rightsquigarrow\ enumFromThenTo\ e_1\ e_2\ e_3
\end{array}
$$

Concatenación de listas

```
Main> [1,2,3] ++ [2,7]
[1,2,3,2,7] :: [Integer]
```

- Definición de $+\!\!+$ atendiendo a la forma del primer argumento:

```
infixr 5 ++

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

$[1,2,3] +\!\!+ [2,7]$
$\rightsquigarrow$! sintaxis de listas
$(1 : (2 : (3 : []))) +\!\!+ [2,7]$
$\Longrightarrow$! segunda ecuación de $(+\!\!+)$
$1 : ((2 : (3 : [])) +\!\!+ [2,7])$
$\Longrightarrow$! segunda ecuación de $(+\!\!+)$
$1 : (2 : ((3 : []) +\!\!+ [2,7]))$
$\Longrightarrow$! segunda ecuación de $(+\!\!+)$
$1 : (2 : (3 : ([] +\!\!+ [2,7])))$
$\Longrightarrow$! primera ecuación de $(+\!\!+)$
$1 : (2 : (3 : [2,7]))$
$\rightsquigarrow$! sintaxis de listas
$[1,2,3,2,7]$

- Propiedades de $(+\!\!+)$:

[] es neutro a derecha:

$$\forall\, xs \;::\; [a] \;\centerdot\; xs +\!\!+ [\,] \;=\; xs$$

y a izquierda:

$$\forall\, ys \;::\; [a] \;\centerdot\; [\,] +\!\!+ ys \;=\; ys$$

Asociatividad:

$$\forall\, xs,\; ys,\; zs \;::\; [a] \;\centerdot\; (xs +\!\!+ ys) +\!\!+ zs \;=\; xs +\!\!+ (ys +\!\!+ zs)$$

- Resulta más eficiente declarar el operador asociativo a la derecha.
- $concat$ toma una lista de listas y devuelve como resultado la lista concatenación de todas ellas:

```
Prelude> concat [ [1,2], [3,4,5], [6] ]
[1,2,3,4,5,6] :: [Integer]
```

```
concat           :: [[a]] -> [a]
concat []        = []
concat (xs : xss) = xs ++ concat xss
```

- Ejercicio.- La función $reverse$ invierte el orden de los elementos en una lista:

```
Prelude> reverse [1,2,3]
[3,2,1] :: [Integer]
```

Define esta función.

Inducción sobre listas

Las listas son estructuras de datos inductivas, por lo hay un principio de inducción para ellas:

Principio de inducción para listas finitas

$$\forall xs :: [a] \;\centerdot\; Pxs \Leftrightarrow \left\{ \begin{array}{l} P[\,] \\ \wedge \\ \forall xs :: [a], \forall x :: a \;\centerdot\; (Pxs \Rightarrow P(x : xs)) \end{array} \right.$$

- Usando el principio anterior, demostraremos la siguiente propiedad:

$(+\!\!+)$ distribuye con $length$ mediante $(+)$

$$\forall\, xs,\; ys \;::\; [a] \;\centerdot\; length\;(xs +\!\!+ ys) \;=\; length\; xs \;+ length\; ys$$

donde

$$\begin{array}{ll} length & :: [a] \to Int \\ length\;[\,] & = 0 \\ length\;(\_ : xs) & = 1 \;+\; length\; xs \end{array}$$

y

$$\begin{array}{lll} (+\!\!+) & & :: [a] \to [a] \to [a] \\ [\,] & +\!\!+\; ys & = ys \\ (x : xs) & +\!\!+\; ys & = x \;:\; (xs +\!\!+ ys) \end{array}$$

Realizaremos la inducción sobre la variable xs por lo que la propiedad a probar es

$$P(xs) \;=\; \forall\, ys \;::\; [a] \;\centerdot\; length\;(xs +\!\!+ ys) \;=\; length\; xs \;+\; length\; ys$$

- CASO BASE: $P\ [\,]$; o sea:

$$\forall\ ys\ ::\ [a]\ \bullet\ length\ ([\,] +\!\!+\ ys)\ =\ length\ [\,]\ +\ length\ ys$$

- PASO INDUCTIVO: $\forall\ xs\ ::\ [a], \forall\ x\ ::\ a\ \bullet P\ xs\ \Rightarrow\ P\ (x : xs)$; o sea:

$$\begin{array}{l}\forall\ xs\ ::\ [a], \forall\ x\ ::\ a\ \bullet \\ \quad \forall\ ys\ ::\ [a]\ \bullet\ length\ (xs +\!\!+\ ys)\ =\ length\ xs\ +\ length\ ys \\ \quad \Rightarrow \\ \quad \forall\ ys\ ::\ [a]\ \bullet\ length\ ((x : xs) +\!\!+\ ys)\ =\ length\ (x : xs)\ +\ length\ ys\end{array}$$

demostración del caso base:

$$\begin{array}{ll} & length\ (\underline{[\,] +\!\!+ ys}) \\ \equiv! & \text{por definición de } (+\!\!+) \\ & length\ ys \end{array}$$

$$\begin{array}{ll} & \underline{length\ [\,]} + length\ ys \\ \equiv! & \text{por definición de } length \\ & \underline{0 + length\ ys} \\ \equiv! & \text{aritmética} \\ & length\ ys \end{array}$$

demostración del paso inductivo:

$$\begin{array}{ll} & length\ (\underline{(x : xs) +\!\!+ ys}) \\ \equiv! & \text{por definición de } (+\!\!+) \\ & \underline{length\ (x : (xs +\!\!+ ys))} \\ \equiv! & \text{por definición de } length \\ & 1 + \underline{length\ (xs +\!\!+ ys)} \\ \equiv! & \text{por HI} \\ & 1 + (length\ xs + length\ ys) \end{array}$$

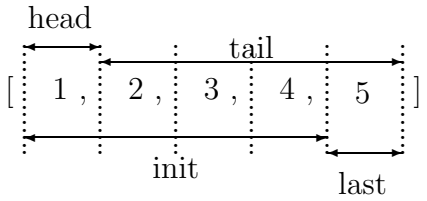
$$\begin{array}{ll} & \underline{length\ (x : xs)} + length\ ys \\ \equiv! & \text{por definición de } length \\ & \underline{(1 + length\ xs) + length\ ys} \\ \equiv! & \text{por asociatividad de } + \\ & 1 + (length\ xs + length\ ys) \end{array}$$

Selectores

Permiten extraer parte de una lista.

$$head, last :: [a] \to a$$
$$head\ (x : \_) = x$$

$$tail, init :: [a] \to [a]$$
$$tail\ (\_ : xs) = xs$$




```
Prelude> head [1..5]
1 :: Integer
Prelude> tail [1..5]
[2,3,4,5] :: [Integer]
Prelude> last [1..5]
5 :: Integer
Prelude> init [1..5]
[1,2,3,4] :: [Integer]
```

Relación entre head y tail

$$\forall\ ls :: [a] \bullet head\ ls : tail\ ls = ls$$

- La función *take*:

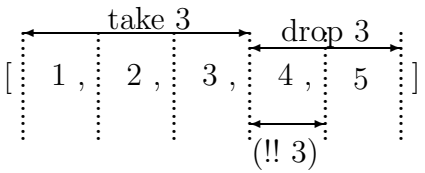
```
Prelude> take 3 [1..5]
[1,2,3] :: [Integer]
Prelude> take 10 [1..5]
[1,2,3,4,5] :: [Integer]
```

```
take                    :: Int -> [a] -> [a]
take 0 _                = []
take _ []               = []
take n (x:xs) | n > 0   = x : take (n-1) xs
take _ _                = error "Prelude.take :
                            negative argument"
```

- La función *drop*:

```
Prelude> drop 3 [1..5]
[4,5] :: [Integer]
Prelude> drop 10 [1..5]
[] :: [Integer]
```

- Ejercicio.- Define la función *drop*.



- *splitAt* combina los resultados de *take* y *drop*:

```
Prelude> splitAt 3 [1..5]
([1,2,3],[4,5]) :: ([Integer],[Integer])
Prelude> splitAt 10 [1..5]
([1,2,3,4,5],[]) :: ([Integer],[Integer])
```

- El operador (!!) :

```
Prelude> [1..5] !! 3
4 :: Integer
Prelude> [1..5] !! 10
Program error : Prelude.!! : index too large
```

```
infixl 9 !!

(!!)                 :: [a] → Int → a
(x : _)  !! 0        = x
(_ : xs) !! n | n > 0 = xs !! (n − 1)
(_ : _)  !! _        = error "Prelude.!! : negative index"
[]       !! _        = error "Prelude.!! : index too large"
```

Emparejando listas

- *zipWith*

```
zipWith :: (a → b → c) → [a] → [b] → [c]

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

```
Prelude> zipWith (+) [1..3] [10..12]
[11,13,15] :: [Integer]
Prelude> zipWith (+) [1..3] [10..]
[11,13,15] :: [Integer]
```

- *zip*

```
zip :: [a] → [b] → [(a, b)]
zip = zipWith (λ a b → (a, b))
```

```
Prelude> zip [1..3] ['a' .. 'c']
[(1,'a'),(2,'b'),(3,'c')] :: [(Integer, Char)]
```

- La función *unzip* realiza el proceso inverso:

```
unzip :: [(a, b)] → ([a], [b])
unzip []           = ([], [])
unzip ((a, b) : xs) = (a : as, b : bs)
  where
    (as, bs) = unzip xs
```

Aplicando una función a los elementos de una lista

```
listaAlCuadrado          :: [Integer] → [Integer]
listaAlCuadrado []       = []
listaAlCuadrado (x : xs) = alCuadrado x  :  listaAlCuadrado xs
  where
    alCuadrado = (↑ 2)
```

```
Main> listaAlCuadrado [1..10]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100] :: [Integer]
```

```
listaAMayúsculas          :: [Char] → [Char]
listaAMayúsculas []       = []
listaAMayúsculas (x : xs) = toUpper x  :  listaAMayúsculas xs
```

```
Main> listaAMayúsculas "hola"
"HOLA" :: [Char]
```

- El combinador map:

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

```
Prelude> map (↑ 2) [1..10]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100] :: [Integer]
```

```
Prelude> map toUpper "hola"
"HOLA" :: [Char]
```

```
Prelude> map ord "hola"
[104, 111, 108, 97] :: [Int]
```

- Ejercicio.- Demuestra las siguientes propiedades:

map distribuye con $(.)$

$$\forall\, g :: a \to b,\ f :: b \to c \,\bullet\, map\,(f.g) = map\,f\,.\,map\,g$$

Relación entre map e id

$$map\ id = id$$

map no modifica la longitud de una lista

$$\forall\, f :: a \to b \,\bullet\, length\,.\,map\,f = length$$

Filtros

- Un *filtro* permite seleccionar los elementos de una lista que cumplen cierta propiedad.

```
Prelude> filter even [1..10]
[2,4,6,8,10] :: [Integer]
Prelude> filter (> 'g') "me gustan las listas"
"mustnlslists" :: [Char]
```

```
filter              :: (a → Bool) → [a] → [a]
filter p []         = []
filter p (x : xs)
  | p x             = x : filter p xs
  | otherwise       = filter p xs
```

- La función *takeWhile*:

```
Prelude> filter even [2,4,8,9,10,11,12]
[2,4,8,10,12] :: [Integer]
Prelude> takeWhile even [2,4,8,9,10,11,12]
[2,4,8] :: [Integer]
```

```
Prelude> takeWhile (< 100) (map (↑ 2) [0..])
[0,1,4,9,16,25,36,49,64,81] :: [Integer]
```

```
takeWhile              :: (a → Bool) → [a] → [a]
takeWhile p []         = []
takeWhile p (x : xs)
  | p x                = x : takeWhile p xs
  | otherwise          = []
```

- La función *dropWhile*:

```
Prelude> dropWhile even [2,4,8,9,10,11,12]
[9,10,11,12] :: [Integer]
```

- Ejercicio.- Define la función *dropWhile*.

filter no aumenta la longitud de una lista

$$\forall\, xs :: [a],\ p :: a \to Bool \ .\ length\ (filter\ p\ xs) \leq length\ xs$$

Listas por comprensión

- HASKELL permite definir listas mediante una notación por comprensión:

$$\{x \mid x \in \mathcal{N}, x \; es \; par\}$$

```
PRELUDE> [ x | x ← [0..], even x]
[2, 4, 6, 8, 10, 12, 14, 16{Interrupted!}
```

- La sintaxis de una lista por comprensión en HASKELL es

$$[\; expresión \mid cualificador_1,\; cualificador_2,\; \ldots,\; cualificador_n \;]$$

donde cada cualificador puede ser:

- Un *generador*: expresión que genera una lista.
- Una *guarda*: expresión booleana.
- Una *definición local*: se usan para definir elementos locales a la expresión.

```
PRELUDE> [2 * x | x ← [1..5]]
[2,4,6,8,10] :: [Integer]
PRELUDE> [2 * x | x ← [1..5], even x]
[4,8] :: [Integer]
```

```
PRELUDE> [x + 1 | let xs = [1..5], x ← xs]
[2,3,4,5,6] :: [Integer]
```

```
map      :: (a → b) → [a] → [b]
map f xs = [ f x | x ← xs]
```

```
filter      :: (a → Bool) → [a] → [a]
filter p xs = [ x | x ← xs, p x]
```

- Si se introduce más de un generador, los que aparecen a la derecha cambian más rápido:

```
PRELUDE> [ (x, y) | x ← [1..3], y ← ['a', 'b'] ]
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')] :: [(Integer, Char)]
PRELUDE> map (λx → map (λy → (x, y)) ['a', 'b']) [1..3]
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b'), (3, 'a'), (3, 'b')] :: [(Integer, Char)]
```

- Los divisores de un número:

```
divideA        :: Integer → Integer → Bool
d `divideA` n  = n `mod` d == 0
divisores      :: Integer → [Integer]
divisores n    = [ x | x ← [1..n], x `divideA` n ]
```

- El máximo común divisor de dos números:

```
mcd      :: Integer → Integer → Integer
mcd a b  = maximum [ x | x ← divisores a, x `divideA` b ]
```

- La lista infinita de los números primos:

```
esPrimo    :: Integer → Bool
esPrimo n  = divisores n == [1, n]
losPrimos  :: [Integer]
losPrimos  = [ n | n ← [2..], esPrimo n ]
MAIN> take 10 losPrimos
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29] :: [Integer]
MAIN> length (takeWhile (< 100) losPrimos)
25 :: Int
MAIN> head (dropWhile (≤ 1000) losPrimos)
1009 :: Integer
```

- Ejercicio.- Escribe una expresión que muestre los primos menores que 1000 que acaban en tres.
- Ejercicio.- Define una función

$$primeroQue :: (a \to Bool) \to [a] \to a$$

que devuelva el primer elemento de una lista (posiblemente infinita) que verifique un predicado.

```
Main> primeroQue (> 1000) losPrimos
1009 :: Integer
```

- Ejercicio.- Un número es perfecto si coincide con la suma de sus divisores (exceptuando el propio número).
  - $6 = 1 + 2 + 3$. Escribe una función que devuelva la lista de los números perfectos.

- Definiciones locales: se introducen con la palabra **let**.
  - El ámbito de la definición introducida abarca todos los cualificadores a la derecha de ésta y la expresión a la izquierda de |.

```
Prelude> [ n | n ← [1..10], let cuadrado = n * n, even cuadrado]
[2, 4, 6, 8, 10] :: [Integer]
```

- Las *ternas pitagóricas*:

```
ternasPitHasta :: Integer → [(Integer, Integer, Integer)]

ternasPitHasta n
  = [ (x, y, z) | let ns = [1..n],
                  x ← ns, y ← ns, z ← ns,
                  x ↑ 2 + y ↑ 2 == z ↑ 2 ]
```

- La siguiente función comprueba si una operación binaria f es conmutativa sobre un conjunto representado por la lista ds:

```
conmuta      :: (a → a → b) → [a] → Bool
conmuta f ds = and [ f a b == f b a | a ← ds, b ← ds ]
```

donde la función predefinida and comprueba si todos los elementos de una lista de booleanos son ciertos.

```
Main> conmuta (+) [1..10]
True :: Bool
Main> conmuta (&&) [True, False]
True :: Bool
```

Semántica de listas por comprensión

$$
\begin{array}{lll}
(1) & [\ e \mid\ ] & \rightsquigarrow\ [\ e\ ] \\
(2) & [\ e \mid b,\ Q\ ] & \rightsquigarrow\ \textbf{if}\ \ b\ \textbf{then}\ \ [\ e \mid Q\ ]\ \textbf{else}\ \ [\,] \\
(3) & [\ e \mid p \leftarrow\ l,\ Q\ ] & \rightsquigarrow\ \textbf{let} \\
 & & \quad ok\ p\ =\ [\ e \mid Q\ ] \\
 & & \quad ok\ \_\ =\ [\,] \\
 & & \textbf{in} \\
 & & \quad concat\ (map\ ok\ l) \\
(4) & [\ e \mid \textbf{let}\ decls,\ Q\ ] & \rightsquigarrow\ \textbf{let} \\
 & & \quad decls \\
 & & \textbf{in} \\
 & & \quad [\ e \mid Q\ ]
\end{array}
$$

$[2 * x | x \leftarrow [1..3], even\ x]$

$\rightsquigarrow$! por regla (3)

```
let
  ok x = [2 * x | even x]
  ok _ = []
in
  concat (map ok [1..3])
```

$\rightsquigarrow$! por regla (2)

```
let
  ok x = if even x then [2 * x | ] else []
  ok _ = []
in
  concat (map ok [1..3])
```

$\rightsquigarrow$! por regla (1)

```
let
  ok x = if even x then [2 * x] else []
  ok _ = []
in
  concat (map ok [1..3])
```

Plegado de listas

- Las funciones de plegado sobre listas capturan el comportamiento de las funciones recursivas que recorren todos los elementos de la lista para calcular un resultado.

$foldr$

- Si el argumento es la lista vacía, se devuelve cierto valor.
- En otro caso, se opera, mediante cierta función u operador, la cabeza de la lista y una llamada recursiva con la cola de la lista.

$$
\begin{array}{lll}
sumar & :: & [Integer] \to Integer \\
sumar\ [\,] & = & 0 \\
sumar\ (x : xs) & = & x + sumar\ xs
\end{array}
\qquad
\begin{array}{lll}
concat & :: & [[a]] \to [a] \\
concat\ [\,] & = & [\,] \\
concat\ (xs : xss) & = & xs +\!\!+ concat\ xss
\end{array}
$$

$$
\begin{array}{lll}
fun\ [\,] & = & \boxed{z} \\
fun\ (x : xs) & = & x\ \boxed{\text{`}f\text{`}}\ (fun\ xs)
\end{array}
$$

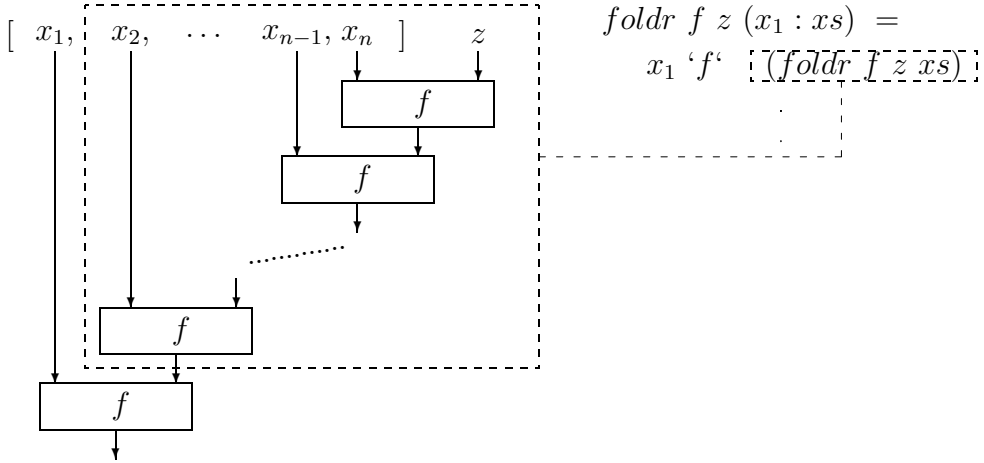
- La función $foldr$

$$
\begin{array}{lll}
foldr & :: & (a \to b \to b) \to b \to [a] \to b \\
foldr\ f\ z\ [\,] & = & z \\
foldr\ f\ z\ (x : xs) & = & x\ \text{`}f\text{`}\ (foldr\ f\ z\ xs)
\end{array}
$$

$$
\begin{array}{lll}
sumar & :: & [Integer] \to Integer \\
sumar & = & foldr\ (+)\ 0
\end{array}
\qquad
\begin{array}{lll}
concat & :: & [[a]] \to [a] \\
concat & = & foldr\ (+\!\!+)\ [\,]
\end{array}
$$

Comportamiento de foldr

$$foldr\ f\ z\ (x_1 : (x_2 : (\cdots : (x_n : [\,])))) \Longrightarrow x_1\ `f`\ (x_2\ `f`\ (\ldots\ `f`\ (x_n\ `f`\ z)))$$



$foldr\ (+)\ 0\ [1,2,3]$
$\rightsquigarrow$! sintaxis de listas
$foldr\ (+)\ 0\ (1 : (2 : (3 : [\,])))$
$\Longrightarrow$! primera ecuación de $foldr$
$1 + (foldr\ (+)\ 0\ (2 : (3 : [\,])))$
$\Longrightarrow$! primera ecuación de $foldr$
$1 + (2 + (foldr\ (+)\ 0\ (3 : [\,])))$
$\Longrightarrow$! primera ecuación de $foldr$
$1 + (2 + (3 + (foldr\ (+)\ 0\ [\,])))$
$\Longrightarrow$! segunda ecuación de $foldr$
$1 + (2 + (3 + 0))$
$\Longrightarrow$! definición de $(+)$ tres veces
6

- Mas ejemplos usando $foldr$

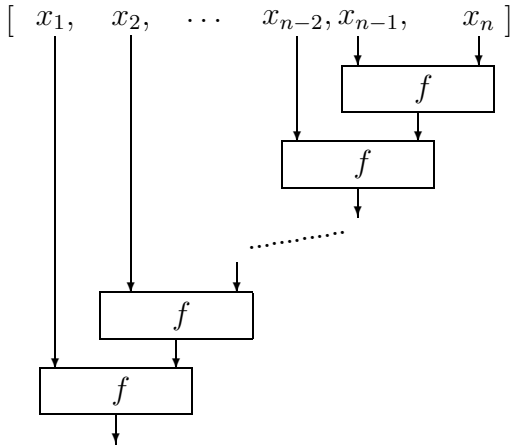
$$length :: [a] \to Int$$
$$length = foldr\ (\lambda x\ n \to n + 1)\ 0$$

$$reverse :: [a] \to [a]$$
$$reverse = foldr\ (\lambda x\ xs \to xs +\!\!+ [x])\ [\,]$$

- Ejercicio.- Define:

$$\begin{aligned}
and &:: [Bool] \to Bool \\
or &:: [Bool] \to Bool \\
all &:: (a \to Bool) \to [a] \to Bool \\
any &:: (a \to Bool) \to [a] \to Bool \\
map &:: (a \to b) \to [a] \to [b] \\
filter &:: (a \to Bool) \to [a] \to [a]
\end{aligned}$$

- Ejercicio.- Define la función $foldr1$



- Ejercicio.- Define usando esta función las funciones $maximum$ y $minimun$.

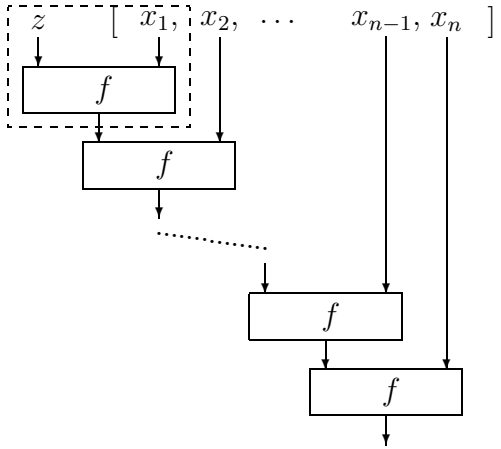
$foldl$

- Similar a $foldr$. Realiza el plegado de la lista de izquierda a derecha.

$$
\begin{aligned}
&foldl && :: (b \to a \to b) \to b \to [a] \to b \\
&foldl\ f\ z\ [\,] && = z \\
&foldl\ f\ z\ (x : xs) && = foldl\ f\ (f\ z\ x)\ xs
\end{aligned}
$$

Comportamiento de foldl

$$foldl\ f\ z\ [x_1, x_2, \ldots, x_n] \Longrightarrow ((((z\ `f`\ x_1)\ `f`\ x_2) \ldots\ `f`\ x_{n-1})\ `f`\ x_n)$$



- La definición de $reverse$ con $foldl$

$$
\begin{aligned}
&reverse && :: [a] \to [a] \\
&reverse && = foldl\ (flip\ (:))\ [\,]
\end{aligned}
$$

es más eficiente que la definición con $foldr$.

Los resultados de $foldr\ f\ z\ xs$ y $foldl\ f\ z\ xs$ coinciden si se cumplen las siguientes tres condiciones:

- f es asociativa: $(a\ `f`\ b)\ `f`\ c \equiv a\ `f`\ (b\ `f`\ c)$.
- z es elemento neutro de f: $\forall\ x\ .\ f\ x\ z \equiv f\ z\ x \equiv x$.
- xs es una lista finita

Cuando falla alguna de las condiciones anteriores, los resultados de $foldr$ y $foldl$ pueden no coincidir:

```
Prelude> foldr (-) 0 [1..5]
3 :: Integer
Prelude> foldl (-) 0 [1..5]
-15 :: Integer
```

```
Prelude> foldr (λp q → False) False [1..]
False :: Bool
Prelude> foldl (λp q → False) False [1..]
...
```

- Ejercicio.- Define la función $foldl1$, que se comporta como $foldr1$ pero plegando de izquierda a derecha.
- Ejercicio.- Define las funciones

$$scanl :: (b \to a \to b) \to b \to [a] \to [b]$$
$$scanr :: (a \to b \to b) \to b \to [a] \to [b]$$

devuelven una lista con los resultados intermedios de $foldl$ y $foldr$ respectivamente:

$$\begin{array}{c} Comportamiento\ de\ scanl \\ scanl\ f\ z\ [x_1, x_2, \ldots, x_n] \\ \Longrightarrow \\ [z\ `f`\ x_1, (z\ `f`\ x_1)\ `f`\ x_2, \ldots, (((z\ `f`\ x_1)\ `f`\ x_2) \ldots\ `f`\ x_n)] \end{array}$$

- Ejercicio.- Escribe una expresión que devuelva la lista $[0!, 1!, 2!, \ldots, 10!]$ usando $scan?$.

Ordenación de listas

- Una lista $[x_1,\ x_2,\ \ldots\ x_n]$ está ordenada ascendentemente si $x_1 \leq x_2 \leq \ldots \leq x_{n-1} \leq x_n$.
- Ejercicio.- Define la función

$$ordenadaAsc :: Ord\ a \Rightarrow [a] \rightarrow Bool$$

que compruebe si una lista está ordenada ascendentemente:

```
Main> ordenadaAsc [1,4,4,7,10]
True :: Bool
Main> ordenadaAsc [1,9,3]
False :: Bool
```

- Ejercicio.- Generaliza la función anterior a la siguiente:

$$ordenadaPor :: (a \rightarrow a \rightarrow Bool) \rightarrow [a] \rightarrow Bool$$

(el primer argumento indica el orden)

```
Main> ordenadaPor (≥) [9,9,8]
True :: Bool
Main> ordenadaPor (>) [9,9,8]
False :: Bool
```

Ordenación por inserción

- La función *insertar*:

```
Main> insertar 4 [1,3,7]
[1,3,4,7] :: [Integer]
```

```
insertar              :: Ord a => a -> [a] -> [a]
insertar x []         = [x]
insertar x (y : ys)
  | x ≤ y             = x : y : ys
  | otherwise         = y : insertar x ys
```

- Para ordenar se parte de una lista vacía

1. $insertar\ 9\ [\,] \Longrightarrow [9]$
2. $insertar\ 3\ [9] \Longrightarrow [3,9]$
3. $insertar\ 7\ [3,9] \Longrightarrow [3,7,9]$

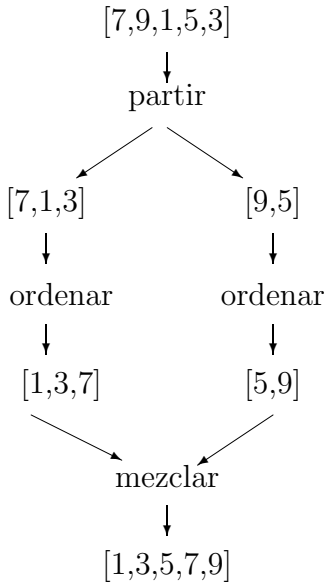
$$insertar\ x_1\ (insertar\ x_2\ \ldots\ (insertar\ x_n\ [\,]))$$

```
ordenarIns :: Ord a => [a] -> [a]
ordenarIns = foldr insertar []
```

```
Main> ordenarIns [7,4,3,6,7]
[3,4,6,7,7] :: [Integer]
```

Ordenación por mezcla

- Se trata de un método del tipo *divide y vencerás*. Ésta es una estrategia general para la resolución de problemas que consta de dos fases:



✓ Para la primera fase:

$$
\begin{array}{lll}
partir & :: [a] \to ([a],[a]) & \\
partir\ [\,] & = ([\,],[\,]) & \\
partir\ [x] & = ([x],[\,]) & \\
partir\ (x:y:zs) & = (x:xs,\ y:ys) & \\
\quad \textbf{where} & & \\
\quad (xs,\ ys) & = partir\ zs &
\end{array}
$$

✓ Para la segunda fase:

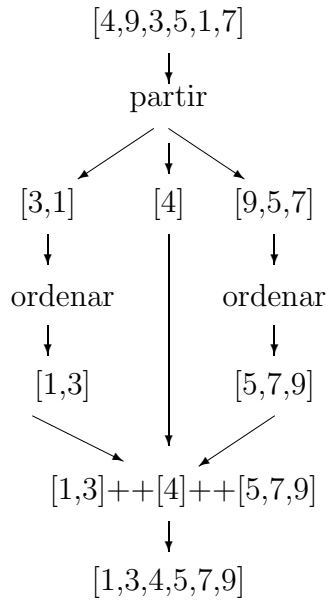
$$
\begin{array}{llll}
mezclar & :: Ord\ a \Rightarrow [a] \to [a] \to [a] & & \\
mezclar & [\,] & ly & = ly \\
mezclar & lx & [\,] & = lx \\
mezclar & lx@(x:xs) & ly@(y:ys) & \\
\quad | & x \leq y & & = x\ :\ mezclar\ xs\ ly \\
\quad | & otherwise & & = y\ :\ mezclar\ lx\ ys
\end{array}
$$

✓ Ordenando por mezcla:

$$
\begin{array}{ll}
ordenarMezcla & :: Ord\ a \Rightarrow [a] \to [a] \\
ordenarMezcla\ [\,] & = [\,] \\
ordenarMezcla\ [x] & = [x] \\
ordenarMezcla\ xs & = mezclar\ (ordenarMezcla\ ys) \\
 & \qquad\quad (ordenarMezcla\ zs) \\
\quad \textbf{where} & \\
\quad (ys, zs) & = partir\ xs
\end{array}
$$

Ordenación rápida de Hoare

- También del tipo divide y vencerás.



- Tomar el primer elemento de la lista a ordenar (*pivote*). Dividir la cola en menores al pivote y resto. Ordenar cada una de estas listas.

- Concatenar los resultados obtenidos.

$$
\begin{aligned}
&quickSort\ [\,] = [\,]\\
&quickSort\ (p : xs)\\
&\quad = quickSort\ menores +\!\!+ [p] +\!\!+ quickSort\ mayores\\
&\quad \textbf{where}\\
&\qquad menores = [\, x \mid x \leftarrow xs,\ x < p \,]\\
&\qquad mayores = [\, x \mid x \leftarrow xs,\ x \geq p \,]
\end{aligned}
$$

- Define una función $partirPor$ que tome una condición y una lista y la parta, dando una única pasada a la lista, en aquellos elementos que cumplen la condición y los que no la cumplen:

$$
\begin{aligned}
&\text{Main>}\ partirPor\ even\ [1,3,2,7,11,8,10]\\
&([2,8,10],[1,3,7,11])\ ::\ ([Integer],\ [Integer])
\end{aligned}
$$

- Ejercicio.- Mejora la definición de la función $quickSort$ usando $partirPor$.

Funciones combinatorias

• Una *función combinatoria* es una función polimórfica que toma una lista y devuelve una lista de listas.

Los prefijos de una lista

$$
\begin{array}{lll}
inits\ [\,] & \Longrightarrow & [\,[\,]\,] \\
inits\ [\,x\,] & \Longrightarrow & [\,[\,], [\,x\,]\,] \\
inits\ [\,y,x\,] & \Longrightarrow & [\,[\,], [\,y\,], [\,y,x\,]\,] \\
inits\ [\,z,y,x\,] & \Longrightarrow & [\,[\,], [\,z\,], [\,z,y\,], [\,z,y,x\,]\,] \\
\ldots &&
\end{array}
$$

La definición en HASKELL es:

$$
\begin{array}{lll}
inits & :: & [a] \to [[a]] \\
inits\ [\,] & = & [\,[\,]\,] \\
inits\ (x:xs) & = & [\,] \;:\; map\ (x:)\ (inits\ xs)
\end{array}
$$

• Ejercicio.- Define una función $tails$ que devuelva los segmentos finales de una lista:

```
MAIN> tails [1,2,3]
[[1,2,3],[2,3],[3],[]] :: [[Integer]]
MAIN> tails ([] :: [Integer])
[[]] :: [[Integer]]
```

Los segmentos de datos consecutivos de una lista

$$
\begin{array}{lll}
segs\ [\,] & \Longrightarrow & [\,[\,]\,] \\
segs\ [\,x\,] & \Longrightarrow & [\,[\,], [\,x\,]\,] \\
segs\ [\,y,x\,] & \Longrightarrow & [\,[\,], [\,y\,], [\,x\,], [\,y,x\,]\,] \\
segs\ [\,z,y,x\,] & \Longrightarrow & \\
\multicolumn{3}{l}{\qquad [\,[\,], [\,z\,], [\,y\,], [\,x\,], [\,z,y\,], [\,y,x\,], [\,z,y,x\,]\,]} \\
\ldots &&
\end{array}
$$

reordenando

$$
\begin{array}{lll}
segs\ [\,] & \Longrightarrow & [\,[\,]\,] \\
segs\ [\,x\,] & \Longrightarrow & [\,[\,], [\,x\,]\,] \\
segs\ [\,y,x\,] & \Longrightarrow & [\,[\,], [\,x\,], [\,y\,], [\,y,x\,]\,] \\
segs\ [\,z,y,x\,] & \Longrightarrow & \\
\multicolumn{3}{l}{\qquad [\,[\,], [\,x\,], [\,y\,], [\,y,x\,], [\,z\,], [\,z,y\,], [\,z,y,x\,]\,]} \\
\ldots &&
\end{array}
$$

$$
\begin{array}{lll}
segs & :: & [a] \to [[a]] \\
segs\ [\,] & = & [\,[\,]\,] \\
segs\ ls@(x:xs) & = & segs\ xs +\!\!+ tail\ (inits\ ls)
\end{array}
$$

• Ejercicio.- Define una función $partes$ que devuelva todos los subconjuntos de un conjunto.

$$
\begin{array}{l}
partes\ [\,1,\ 2,\ 3\,] \\
\qquad \Longrightarrow\ [\,[\,], [1], [2], [3], [1,2], [2,3], [1,3], [1,2,3]\,]
\end{array}
$$

Permutaciones de una lista

$$
\begin{array}{lll}
perms\ [\,] & \Longrightarrow & [\,[\,]\,] \\
perms\ [\,x\,] & \Longrightarrow & [\,[\,x\,]\,] \\
perms\ [\,y,x\,] & \Longrightarrow & [\,[\,y,x\,],\ [x,y]\,] \\
perms\ [\,z,y,x\,] & \Longrightarrow & [\,[\,z,y,x\,],\ [\,y,z,x\,],\ [\,y,x,z\,], \\
 & & \quad [\,z,x,y\,],\ [\,x,z,y\,],\ [\,x,y,z\,]\,]
\end{array}
$$

$\ldots$

- Hay tres modos de intercalar z en la lista $[\,y,x\,]$:
- Las otras tres permutaciones se obtienen intercalando z en $[\,x,y\,]$.

$$
\begin{array}{l}
\textsc{Main}{>}\ intercala\ 3\ [1,2] \\
[\,[\,3,1,2\,],\ [\,1,3,2\,],\ [\,1,2,3\,]\,]\ ::\ [[Integer]]
\end{array}
$$

$$
\begin{array}{lll}
intercala\ 1\ [\,] & \Longrightarrow & [\,[\,1\,]\,] \\
intercala\ 1\ [4] & \Longrightarrow & [\,[\,1,4\,],\ [\,4,1\,]\,] \\
intercala\ 1\ [3,4] & \Longrightarrow & [\,[\,1,3,4\,],\ [\,3,1,4\,],\ [\,3,4,1\,]\,]
\end{array}
$$

$\ldots$

$$
\begin{array}{l}
\textsc{Main}{>}\ intercala\ 1\ [3,4] \\
[[1,3,4],[3,1,4],[3,4,1]]\ ::\ [[Integer]] \\
\textsc{Main}{>}\ map\ (2:)\ (intercala\ 1\ [3,4]) \\
[[2,1,3,4],[2,3,1,4],[2,3,4,1]]\ ::\ [[Integer]] \\
\textsc{Main}{>}\ (1:[2,3,4])\ :\ map\ (2:)\ (intercala\ 1\ [3,4]) \\
[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1]]\ ::\ [[Integer]]
\end{array}
$$

$$
\begin{array}{lll}
intercala & :: & a \to [a] \to [\,[a]\,] \\
intercala\ x\ [\,] & = & [\,[x]\,] \\
intercala\ x\ ls@(y:ys) & = & (x:ls)\ :\ map\ (y:)\ (intercala\ x\ ys)
\end{array}
$$

- Partiendo de $perms\ [1,2]$

$$
\begin{array}{l}
\textsc{Main}{>}\ perms\ [1,2] \\
[[1,2],[2,1]]\ ::\ [[Integer]] \\
\textsc{Main}{>}\ intercala\ 0\ [1,2] \\
[[0,1,2],[1,0,2],[1,2,0]]\ ::\ [[Integer]] \\
\textsc{Main}{>}\ intercala\ 0\ [2,1] \\
[[0,2,1],[2,0,1],[2,1,0]]\ ::\ [[Integer]] \\
\textsc{Main}{>}\ map\ (intercala\ 0)\ (perms\ [1,2]) \\
[[[0,1,2],[1,0,2],[1,2,0]],[[0,2,1],[2,0,1],[2,1,0]]] \\
\quad ::\ [[[Integer]]] \\
\textsc{Main}{>}\ concat\ (map\ (intercala\ 0)\ (perms\ [1,2])) \\
[[0,1,2],[1,0,2],[1,2,0],[0,2,1],[2,0,1],[2,1,0]]\ ::\ [[Integer]]
\end{array}
$$

- Ahora es fácil definir $perms$

$$
\begin{array}{l}
perms\ ::\ [a] \to [[a]] \\
perms\ [\,] = [\,[\,]\,] \\
perms\ (x:xs) = concat\ (map\ (intercala\ x)\ (perms\ xs))
\end{array}
$$

Otras funciones predefinidas

- *span p xs* devuelve en una tupla los resultados de *takeWhile p xs* y *dropWhile p xs*.

```
span                  :: (a → Bool) → [a] → ([a],[a])
span p []             = ([],[])
span p xs@(x : xs')
  | p x               = (x : ys, zs)
  | otherwise         = ([], xs)
  where
    (ys, zs) = span p xs'
```

- *break* se comporta como *span* pero con la condición negada:

```
break    :: (a → Bool) → [a] → ([a],[a])
break p  = span (not . p)
```

```
Prelude> span (< 5) [1..10]
([1,2,3,4],[5,6,7,8,9,10]) :: ([Integer],[Integer])
Prelude> break (>= 5) [1..10]
([1,2,3,4],[5,6,7,8,9,10]) :: ([Integer],[Integer])
```

- *words* extrae palabras de una cadena de caracteres:

```
words    :: String → [String]
words s  = case dropWhile isSpace s of
             "" → []
             s' → w : words s''
               where
                 (w, s'') = break isSpace s'
```

```
Prelude> words "hola qué pasa"
["hola", "qué", "pasa"] :: [String]
```

- *lines* extrae las líneas de un texto:

```
lines      :: String → [String]
lines ""   = []
lines s    = let
               (l, s') = break ('\n' ==) s
             in
               l : case s' of
                     []      → []
                     (_ : s'') → lines s''
```

```
Prelude> lines "hola\nqué pasa"
["hola", "qué pasa"] :: [String]
```

- *unwords* y *unlines* realizan los procesos inversos.