GLOSARIO

```
curry :: ((a, b) -> c) -> a -> b -> c -- Devuelve una función currificada
curry f x y = f (x, y)
uncurry :: (a -> b -> c) -> (a, b) -> c -- Devuelve una función no currificada
uncurry f (x, y) = f x y
```

Folds

```
foldr :: (a -> b -> b) -> b -> [a] -> b  -- Fold derecho (asociativo a derecha).
foldl :: (b -> a -> b) -> b -> [a] -> b  -- Fold izquierdo (asociativo a izquierda).
foldr1 :: (a -> a -> a) -> [a] -> a      -- `foldr` en listas no vacías.
foldl1 :: (a -> a -> a) -> [a] -> a      -- `foldl` en listas no vacías.
```

Listas

```
map :: (a -> b) -> [a] -> [b]                    -- Aplica función a cada elemento.
filter :: (a -> Bool) -> [a] -> [a]    -- Filtra elementos que cumplen un predicado.
concat :: [[a]] -> [a]                           -- Concatena listas anidadas.
concatMap :: (a -> [b]) -> [a] -> [b]            -- `concat . map`.
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]    -- Combina listas con una función.
reverse :: [a] -> [a]                            -- Invierte la lista.
```

Cosas sobre listas

```
(++) :: [a] -> [a] -> [a]      -- Concatena listas.
head :: [a] -> a               -- Primer elemento (error si vacía).
tail :: [a] -> [a]             -- Lista sin el primer elemento.
init :: [a] -> [a]             -- Lista sin el último elemento.
last :: [a] -> a               -- Último elemento (error si vacía).
length :: [a] -> Int           -- Longitud de la lista.
null :: [a] -> Bool            -- Verifica si la lista está vacía.
replicate :: Int -> a -> [a]   -- Repite un elemento `n` veces.
take :: Int -> [a] -> [a]      -- Toma los primeros `n` elementos.
drop :: Int -> [a] -> [a]      -- Elimina los primeros `n` elementos.
```

Búsqueda

```
elem :: Eq a => a -> [a] -> Bool   -- Verifica si un elemento está en la lista.
any :: (a -> Bool) -> [a] -> Bool  -- Verifica si algún elemento cumple el predicado.
all :: (a -> Bool) -> [a] -> Bool  -- Verifica si todos cumplen el predicado.
```

Orden y duplicados

```
sort :: Ord a => [a] -> [a]                  -- Ordena una lista.
sortBy :: (a -> a -> Ordering) -> [a] -> [a] -- Ordena con función de comparación.
nub :: Eq a => [a] -> [a]                    -- Elimina elementos duplicados.
union :: Eq a => [a] -> [a] -> [a]           -- Unión de conjuntos (sin duplicados).
intersect :: Eq a => [a] -> [a] -> [a]       -- Intersección de conjuntos.
```

Aritmética, lógica y comparación

```
sum :: Num a => [a] -> a                          -- Suma de elementos.
mod :: Integral a => a -> a -> a                  -- Módulo.
div :: Integral a => a -> a -> a                  -- División entera.
odd :: Integral a => a -> Bool                    -- Verifica si es impar.
even :: Integral a => a -> Bool                   -- Verifica si es par.
and :: [Bool] -> Bool                             -- AND lógico sobre una lista.
or :: [Bool] -> Bool                              -- OR lógico sobre una lista.
not :: Bool -> Bool                               -- Negación lógica.
(==), (/=) :: Eq a => a -> a -> Bool              -- Igualdad/desigualdad.
compare :: Ord a => a -> a -> Ordering            -- Compara (`LT`, `EQ`, `GT`).
comparing :: Ord a => (b -> a) -> b -> b -> Ordering -- Compara usando una función.
max, min :: Ord a => a -> a -> a                  -- Máximo/mínimo entre dos valores.
maximum, minimum :: Ord a => [a] -> a             -- Máximo/mínimo de una lista.
maximumBy, minimumBy :: (a -> a -> Ordering) -> [a] -> a -- Máximo/mínimo con comparador.
```

Esquemas de recursión:

- **Estructural:** permite acceder a los argumentos no recursivos de los constructores, y a los resultados de la recursión para las subestructuras.
- **Primitiva:** como la estructural, pero además permite acceder a las subestructuras.
- **Global:** como la primitiva, pero además permite acceder a los resultados de las recursiones anteriores

Ejemplos:

```
longitud [] = 0
longitud (_:xs) = 1 + longitud xs

insertarOrdenado e [] = [e]
insertarOrdenado e (x:xs) = if e < x then e:x:xs
                                     else x:(insertarOrdenado e xs)

elementosEnPosicionesPares [] = []
elementosEnPosicionesPares (x:xs) = if null xs then [x]
                                  else x:elementosEnPosicionesPares (tail xs)
```

1. La recursión de `longitud` es **estructural**, porque hace recursión sobre la cola de la lista (xs) pero no accede a la cola en sí, ni a resultados de recursiones anteriores.
2. La recursión de `insertarOrdenado` es **primitiva** porque accede directamente a xs (además de hacer recursión), pero no accede a los resultados anteriores.
3. La recursión de `elementosEnPosicionesPares` es **global**, ya que accede a un resultado anterior: el de la recursión sobre la cola de la cola de la lista (es decir tail xs).

Funciones clave:

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

foldl :: (b -> a -> b) -> b -> [a] -> b -- Estructural
foldl _ z []     = z
foldl f z (x:xs) = foldl f (f z x) xs

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

Construcción de folds personalizados:

Un esquema de recursión estructural espera recibir un argumento por cada constructor (para saber qué devolver en cada caso), y además la estructura que va a recorrer.

El tipo de cada argumento va a depender de lo que reciba el constructor correspondiente. (¡Y todos van a devolver lo mismo!)

Si el constructor es recursivo, el argumento correspondiente del fold va a recibir el resultado de cada llamada recursiva.

Un pequeño ejemplo:

```
data AB a = Hoja | Nodo (AB a) a (AB a)
  -- Constructor base: Hoja
  -- Constructor recursivo: Nodo

foldAB :: b -> (b -> a -> b -> b) -> AB a -> b
foldAB casoHoja _ Hoja = casoHoja  -- Para Hoja
foldAB casoHoja f (Nodo izq val der) =
   f (foldAB casoHoja f izq) val (foldAB casoHoja f der)
   -- Para Nodo: recibe resultados recursivos (izq/der) y el valor 'val'
```

Algunos ejercicios:

```
--6-------------------------------------------------------------------------------
sacarUna :: Eq a => a -> [a] -> [a]
sacarUna e = recr (\x xs acc -> if x == e then xs else x:acc) []

insertarOrdenado :: Ord a => a -> [a] -> [a]
insertarOrdenado e = recr (\x xs acc -> if x <= e && e <= head xs then x:e:xs else x:acc) []
--7-------------------------------------------------------------------------------
mapPares :: (a -> b -> c) -> [(a,b)] -> [c]
mapPares f = foldr (\(x,y) acc -> f x y : acc) []

armarPares :: [a] -> [b] -> [(a,b)]
armarPares [] _           = []
armarPares _ []           = []
armarPares (x:xs) (y:ys)  = (x,y): armarPares xs ys

-- con foldr
armarPares' :: [a] -> [b] -> [(a,b)]
armarPares' = foldr (\x acc (y:ys) -> (x,y):acc ys) (const [])

mapDoble :: (a -> b -> c) -> [a] -> [b] -> [c]
mapDoble _ [] _           = []
mapDoble _ _ []           = []
mapDoble f (x:xs) (y:ys)  = f x y : mapDoble f xs ys

--9-------------------------------------------------------------------------------

foldNat :: (Integer -> b -> b) -> b -> Integer -> b
foldNat _ z 0 = z
foldNat f z n = f n (foldNat f z (n-1))

potencia :: Integer -> Integer -> Integer
potencia b = foldNat (\_ acc -> b*acc) 1
--10------------------------------------------------------------------------------

genLista :: a -> (a -> a) -> Integer -> [a]
genLista inicio f len = foldr (\_ g x -> x : g (f x))  (const []) [1..len] inicio

--11------------------------------------------------------------------------------

data Polinomio a = X
                 | Cte a
                 | Suma (Polinomio a) (Polinomio a)
                 | Prod (Polinomio a) (Polinomio a)

evaluar :: Num a => a -> Polinomio a -> a
evaluar e X = e
evaluar e (Cte x) = x
evaluar e (Suma p q) = evaluar e p + evaluar e q
evaluar e (Prod p q) = evaluar e p * evaluar e q

--12------------------------------------------------------------------------------
data AB a = Nil | Bin (AB a) a (AB a)

foldAB :: (b -> a -> b -> b) -> b -> AB a -> b
foldAB _ z Nil = z
foldAB f z (Bin i c r) = f (foldAB f z i) c (foldAB f z r)

recAB :: (AB a -> a -> AB a -> b -> b -> b) -> b -> AB a -> b
recAB _ z Nil = z
recAB f z (Bin i c r) = f i c r (recAB f z i) (recAB f z r)
```

```
esNil :: AB a -> Bool
esNil Nil = True
esNil _ = False

altura :: AB a -> Integer
altura = foldAB (\i _ r -> 1 + max i r) 0

cantNodos :: AB a -> Integer
cantNodos = foldAB (\i _ r -> i+1+r) 0

mejorSegun :: (a -> a -> Bool) -> AB a -> a
mejorSegun f (Bin i c r) = foldAB (\i c r -> mejor f c (mejor f i r)) c (Bin i c r)
  where
    mejor f x y = if f x y then x else y

esABB :: Ord a => AB a -> Bool
esABB = recAB (\i c r recI recR -> all (<= c) (abALista i) && all (>= c) (abALista r) && recI
&& recR) True
  where
    abALista = foldAB (\i c r -> i++[c]++r) []

--15--------------------------------------------------------------------------------

data RT a = Nodo a [RT a]

foldRT :: (a -> [b] -> b) -> RT a -> b
foldRT f (Nodo r hijos) = f r (map (foldRT f) hijos)

hojas :: RT a -> [a]
hojas = foldRT (\r hijos -> if null hijos then [r] else concat hijos)

distancias :: RT a -> [(a,Int)]
distancias rt = zip (rtALista rt) (distanciasAux rt)
  where
    rtALista =      foldRT (\r hijos -> if null hijos then [r] else concat hijos)
    distanciasAux = foldRT (\r hijos -> if null hijos then [0] else map (+1) (concat hijos))

alturaRT :: RT a -> Int
alturaRT = foldRT (\r hijos -> if null hijos then 0 else 1 + maximum hijos)
```