Principales funciones estándar para listas en Haskell

(Las definiciones de estas funciones no son necesariamente las internas del sistema)

```
La concatenación de listas (++) :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)
head :: [\alpha] \rightarrow \alpha aplicada a una lista no vacía devuelve su primer elemento.
head (x:xs) = x
tail :: [\alpha] \rightarrow [\alpha] aplicada a una lista no vacía devuelve la lista sin su primer elemento.
tail(x:xs) = xs
last :: [\alpha] -> \alpha aplicada a una lista no vacía devuelve su último elemento.
last[x] = x
last(x:xs) = last xs
init :: [\alpha] - [\alpha] aplicada a una lista no vacía devuelve la lista sin su último elemento.
init [x] = []
init(x:xs) = x:init xs
null :: [\alpha] ->Bool aplicada a una lista decide si es vacía.
                              NOTA: el tipo inferido es null :: Foldable t => t \alpha \rightarrow Bool
null [] = True
null(x:xs) = False
length :: [\alpha] ->Int aplicada a una lista devuelve su longitud.
                             NOTA: el tipo inferido es length :: Foldable t \Rightarrow t \alpha \rightarrow Int
length [] = 0
length (x:xs) = 1 + length xs
take :: Int->[\alpha]->[\alpha] aplicada a un entero n y una lista xs, devuelve una lista con los n
primeros elementos de xs. Si n es mayor que la longitud de xs, se devuelve xs.
take n xs
   | n <= 0
                        =[]
   |xs == []
                        = []
   otherwise
                        = head xs: take (n-1) (tail xs)
drop :: Int\rightarrow[\alpha]\rightarrow[\alpha] aplicada a un entero n y una lista xs, devuelve la lista con los n
```

primeros elementos de xs quitados. Si n es mayor que la longitud de xs, se devuelve []. drop n xs

splitAt :: Int \rightarrow [α] \rightarrow ([α],[α]) aplicada a un entero **n** y a una lista **xs**, devuelve el par (take n xs, drop n xs). Su definición recursiva es:

```
splitAt n xs  | n <= 0 \qquad = ([], xs) \\ | xs == [] \qquad = ([], []) \\ | otherwise \qquad = (head xs : ys, zs) where (ys,zs) = splitAt (n-1) (tail xs)   reverse :: [\alpha] -> [\alpha] \quad aplicada \ a \ una \ lista \ devuelve \ su \ inversa.   reverse \ [] = [] \\  reverse \ (x:xs) = reverse \ xs ++ [x]
```

- Nota: Se verán definiciones alternativas más eficientes para reverse.

Algunas funciones de orden superior

takeWhile :: $(\alpha -> Bool) -> [\alpha] -> [\alpha]$ aplicada a un predicado p y una lista xs, toma los elementos de xs mientras satisfacen p.

dropWhile :: $(\alpha -> Bool) -> [\alpha] -> [\alpha]$ aplicada a un predicado p y una lista xs, salta los elementos de xs mientras satisfacen p.

map :: $(\alpha -> \beta)->[\alpha]->[\beta]$ aplicada a una función f y a una lista xs, devuelve una lista donde f se ha aplicado a cada elemento de xs.

```
map f[] = []
map f(x:xs) = fx : map fxs
```

- También se puede definir mediante una *lista intensional*: map $f xs = [f x \mid x < -xs]$

filter :: $(\alpha -> Bool) -> [\alpha] -> [\alpha]$ aplicada a un predicado p y una lista xs, devuelve una lista que contiene sólo los elementos que satisfacen p.

- También se puede definir mediante una *lista intensional*: filter p $xs = [x \mid x < -xs, p \mid x]$

La familia de funciones zip:

```
zip :: [\alpha] \rightarrow [\beta] \rightarrow [(\alpha, \beta)] aplicada a dos listas devuelve una lista de pares, formados con
los correspondientes elementos de las listas dadas.
zip [] ys = []
zip(x:xs) = [
zip(x:xs)(y:ys) = (x,y) : zip xs ys
- Ejemplo: zip [0,1,2,3] "type" = [(0,t'),(1,y'),(2,p'),(3,e')]
- Si las listas tienen distinta longitud, el resultado tiene la longitud de la lista más corta.
unzip :: [(\alpha,\beta)] \rightarrow ([\alpha],[\beta]) toma una lista de pares y los separa en dos listas.
Ejemplo: unzip [(0,t'),(1,y'),(2,p'),(3,e')] = ([0,1,2,3],"type")
- Similar a zip pero para <u>tres</u> listas argumento y devolviendo una lista de triples, se tiene
zip3 :: [\alpha] \rightarrow [\beta] \rightarrow [\gamma] \rightarrow [(\alpha, \beta, \gamma)] y la correspondiente unzip3 :: [(\alpha, \beta, \gamma)] \rightarrow ([\alpha], [\beta], [\gamma])
zipWith :: (\alpha -> \beta -> \gamma) -> [\alpha] -> [\beta] -> [\gamma] aplicada a una función f y a dos listas xs e ys,
da como resultado: map (uncurry f) (zip xs ys). Su definición recursiva es:
zipWith f [] ys = []
zipWith f(x:xs)[] = []
zipWith f(x:xs)(y:ys) = f x y : zipWith f xs ys
- Ejemplo: zipWith (+) [2,5,0] [4,8,3] = [6,13,3]
- Similarmente se tiene zipWith3 :: [\alpha -> \beta -> \gamma -> \delta] -> [\alpha] -> [\beta] -> [\gamma] -> [\delta]
La familia de funciones fold:
foldl :: (\beta - > \alpha - > \beta) - > \beta - > [\alpha] - > \beta aplicada a una función binaria f, un valor inicial e y
una lista xs, aplana xs en forma asociativa a izquierdas.
                                    - Idea: foldl (op) e [x1,x2,x3] = (((e op x1) op x2) op x3)
foldl f e = e
fold f = (x:xs) = fold f (f = x) xs
NOTA: el tipo inferido es foldl :: Foldable t => (\beta -> \alpha -> \beta) -> \beta -> t \alpha -> \beta
foldr :: (\alpha -> \beta -> \beta) -> \beta -> [\alpha] -> \beta aplicada a una función binaria f, un valor inicial e y
una lista xs, aplana xs en forma asociativa a derechas.
                                    - Idea: foldr (op) e [x1,x2,x3] = x1 op (x2 op (x3 op e))
foldr f \in [] = e
foldr f e(x:xs) = f x (foldr f e xs)
NOTA: el tipo inferido es foldr:: Foldable t => (\alpha -> \beta -> \beta) -> \beta -> t \alpha -> \beta
foldl1 y foldr1 :: (\alpha -> \alpha -> \alpha) -> [\alpha] -> \alpha son las versiones de foldl y foldr para listas no
vacías (no llevan valor inicial e). Sus definiciones son:
foldl1 f(x:xs) = foldl f x xs
foldr1 f(x:xs) = if null xs then x else f x (foldr1 f xs)
NOTA: el tipo inferido es foldl1, foldr1 :: Foldable t => (\alpha -> \alpha -> \alpha) -> t \alpha -> \alpha
```

Funciones definidas en términos de foldr y foldl:

```
\begin{array}{lll} \textbf{sum} :: & \text{Num } \alpha => [\alpha] -> \alpha & \text{suma los elementos de una lista de números} \\ \textbf{product} :: & \text{Num } \alpha => [\alpha] -> \alpha & \text{multiplica los elementos de una lista de números} \\ \textbf{and} :: & [Bool] -> Bool & \text{conjunción de los elementos de una lista de booleanos} \\ \textbf{or} :: & [Bool] -> Bool & \text{disyunción de los elementos de una lista de booleanos} \\ \textbf{concat} :: & [[\alpha]] -> [\alpha] & \text{concatena los elementos de una lista de listas} \\ \end{array}
```

NOTA: actualmente los tipos inferidos usan t α con Foldable t para generalizar $[\alpha]$

Funciones definidas en términos de foldr1 y foldl:

```
maximum :: Ord \alpha => [\alpha] -> \alpha elemento máximo de una lista no vacía minimum :: Ord \alpha => [\alpha] -> \alpha elemento mínimo de una lista no vacía
```

NOTA: actualmente los tipos inferidos usan t α con Foldable t para generalizar [α]

La familia de funciones scan:

scanl aplicada a un operador binario **op** y un valor inicial **e** y una lista **xs**, devuelve la lista obtenida al aplicar **foldl op e** a cada segmento inicial de xs.

```
- Ejemplo: scanl (+) 0 computa las sumas acumuladas de una lista de números dada scanl (+) 0 [2,4,5,8] = [0,2,6,11,19] (los segmentos iniciales de [2,4,5,8] son [], [2], [2,4], [2,4,5] y [2,4,5,8])
```

Una definición eficiente de scanl es:

```
scanl :: (\beta \rightarrow \alpha \rightarrow \beta) \rightarrow \beta \rightarrow [\alpha] \rightarrow [\beta]
scanl f e [] = [e]
scanl f e (x:xs) = e: scanl f (f e x) xs
```

scanr aplicada a un operador binario **op** y un valor inicial **e** y una lista **xs**, devuelve la lista obtenida al aplicar **foldr op e** a cada segmento final de xs.

```
- Ejemplo: scanr (+) 0 [2,4,5,8] = [19,17,13,8,0] (los segmentos finales de [2,4,5,8] son [2,4,5,8], [4,5,8], [5,8], [8] y [])
```

Una definición eficiente de scanr es:

```
scanr :: (\alpha -> \beta -> \beta) -> \beta -> [\alpha] -> [\beta]
scanr f e [] = [e]
scanr f e (x:xs) = f x (head ys) : ys
where ys = scanr f e xs
```

- También se tienen las versiones scanl1 y scanr1 :: $(\alpha -> \alpha -> \alpha) -> [\alpha] -> [\alpha]$ que se aplican sin el valor inicial e
- *Ejemplo:* scanl1 (+) [2,4,5,8] = [2,6,11,19]