Principales funciones estándar para listas en Haskell

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

La concatenación de listas **(++)** :: [α]–>[α]–>[α]
[] ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)

head :: [α]–> α aplicada a una lista no vacía devuelve su primer elemento.
head (x:xs) = x

tail :: [α]–>[α] aplicada a una lista no vacía devuelve la lista sin su primer elemento.
tail (x:xs) = xs

last :: [α]–>α aplicada a una lista no vacía devuelve su último elemento.
last [x] = x
last (x:xs) = last xs

init :: [α]–>[α] aplicada a una lista no vacía devuelve la lista sin su último elemento.
init [x] = []
init (x:xs) = x:init xs

null :: [α]–>Bool aplicada a una lista decide si es vacía.
NOTA: el tipo inferido es null :: Foldable t => t α -> Bool
null [] = True
null (x:xs) = False

length :: [α]–>Int aplicada a una lista devuelve su longitud.
NOTA: el tipo inferido es length :: Foldable t => t α -> Int
length [] = 0
length (x:xs) = 1+ length xs

take :: Int–>[α]–>[α] 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–>[α]–>[α] 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
    | n <= 0        = xs
    | xs == []      = []
    | otherwise     = drop (n-1) (tail xs)
```

splitAt :: Int–>[α]–>([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        = ([], xs)
    | xs == []      = ([], [])
    | otherwise     = (head xs : ys, zs) where (ys,zs) = splitAt (n-1) (tail xs)
```

reverse :: [α]–>[α] 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 :: (α –>Bool)–>[α]–>[α] aplicada a un predicado p y una lista xs, toma los elementos de xs mientras satisfacen p.

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

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

```
dropWhile p [] = []
dropWhile p (x:xs)
        | p x           = dropWhile p xs
        | otherwise     = x:xs
```

map :: (α –> β)–>[α]–>[β] 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) = f x : map f xs
```

- También se puede definir mediante una *lista intensional:* map f xs = [f x | x<–xs]

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

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

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

La familia de funciones zip:

zip :: [α]–>[β]–>[(α,β)] 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 :: [(α,β)]–>([α],[β]) 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 tres listas argumento y devolviendo una lista de triples, se tiene zip3 :: [α]–>[β]–>[γ]–>[(α,β,γ)] y la correspondiente unzip3 :: [(α,β,γ)]–>([α],[β],[γ])

zipWith :: (α–>β–>γ)–>[α]–>[β]–>[γ] 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 :: [α–>β–>γ–> δ]–> [α]–>[β]–>[γ]–>[δ]

La familia de funciones fold:

foldl :: (β–>α–>β)–>β–>[α]–>β 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
foldl f e (x:xs) = foldl f (f e x) xs
```

NOTA: el tipo inferido es foldl :: Foldable t => (β –> α –> β) –> β –> t α –> β

foldr :: (α–>β–>β)–>β–>[α]–>β 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 e [] = e
foldr f e (x:xs) = f x (foldr f e xs)
```

NOTA: el tipo inferido es foldr :: Foldable t => (α –> β –> β) –> β –> t α –> β

foldl1 y foldr1 :: (α–>α–>α)–>[α]–>α 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 => (α –> α –> α) –> t α –> α

Funciones definidas en términos de foldr y foldl:

sum :: Num α => [α] –> α suma los elementos de una lista de números
product :: Num α => [α] –> α multiplica los elementos de una lista de números
and :: [Bool]–>Bool conjunción de los elementos de una lista de booleanos
or :: [Bool] –>Bool disyunción de los elementos de una lista de booleanos
concat :: [[α]]–>[α] concatena los elementos de una lista de listas

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

Funciones definidas en términos de foldr1 y foldl:

maximum :: Ord α => [α] –> α elemento máximo de una lista no vacía
minimum :: Ord α => [α] –> α 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 :: (β –> α –> β) –> β –> [α] –> [β]
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 :: (α–>β–>β)–>β–>[α]–>[β]
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 :: (α–>α–>α)–>[α]–>[α] que se aplican sin el valor inicial **e**
- *Ejemplo:* scanl1 (+) [2,4,5,8] = [2,6,11,19]