FUNDAMENTOS DE LA COMPUTACIÓN
LABORATORIO 3 - LISTAS

Recuerden agregar las siguientes líneas al principio del programa:

```
{-#LANGUAGE GADTs#-}
{-# OPTIONS_GHC -fno-warn-tabs #-}
```

Luego se debe comenzar el programa Haskell con la declaración del nombre del módulo, que debe ser el mismo que el del archivo (Listas.hs).

```
module Listas where
```

Como las funciones que vamos a definir ya existen en el Preludio de Haskell, debemos incluir las siguientes línea que evitan que nuestras definiciones se superpongan con las primitivas de Haskell:

```
import Prelude hiding (null,length,sum,map,zip,zipWith,filter,and,or,any,all,(++),
reverse,concat,head,tail,last,init,(!!),fst,snd,take,drop,takeWhile,dropWhile,split)
```

Además, importamos el Laboratorio anterior (Naturales.hs), que debe estar en el mismo directorio que este:

```
import Naturales
```

Ej.1. Listas

Defina las siguientes funciones sobre listas:

1) `null :: [a]-> Bool`, que devuelve `True` si una lista está vacía.

2) `length :: [a]-> N`, que calcula la longitud (cantidad de elementos) de una lista.

3) `duplicate :: [a] -> [a]` que recibe una lista y devuelve otra, donde cada aparición de un elemento de la lista original es duplicada.
   Por ejemplo: `duplicate [1,2,3] = [1,1,2,2,3,3]`

4) `sum :: [N]-> N`, que suma todos los elementos de una lista de números.

5) `prod :: [N]-> N`, que multiplica todos los elementos de una lista de números.

6) `map :: (a->b) ->[a]-> [b]` que aplica una función a todos los elementos de una lista.
   Por ejemplo: `map not [True,True,False] = [False, False,True]`.

7) `zipWith :: (a->b->c)-> [a]-> [b]-> [c]`, que construye una lista cuyos elementos se calculan a partir de aplicar la función dada a los elementos de las listas de entrada que ocurren en la misma posición en ambas listas. Si una lista es más larga que otra, no se consideran los elementos sobrantes. Por ejemplo: `zipWith (+) [1,3,5] [2,4,6,8] = [3,7,11]`.

8) `filter :: (a->Bool) ->[a]-> [a]`, que recibe un predicado y una lista y devuelve la lista con los elementos para los cuales el predicado es verdadero.

9) `and :: [Bool]-> Bool`, que calcula la conjunción `(&&)` de una lista de booleanos.

10) `or :: [Bool]-> Bool`, que calcula la disyunción `(||)` de una lista de booleanos.

11) `count :: (a->Bool) ->[a]-> N`, que recibe un predicado y una lista y calcula la cantidad de elementos de una lista para los cuales el predicado es verdadero.

12) `any :: (a->Bool) ->[a]->Bool`, que recibe un predicado y una lista y verifica si existe algún elemento de la lista para el cual el predicado es verdadero.

13) `all :: (a->Bool) ->[a]->Bool` que recibe un predicado y una lista y verifica si el predicado es verdadero para todos los elementos de la lista.

14) Redefina las tres funciones anteriores sin utilizar case, sino funciones definidas anteriormente.

15) `(++) :: [a]-> [a]-> [a]`, que concatena dos listas.

16) `reverse :: [a]-> [a]`, que invierte una lista.
Por ejemplo: `reverse [1,2,3,4] = [4,3,2,1]`.

17) `elem :: Eq a => a-> [a]-> Bool`, que verifica si un elemento pertenece a una lista.

18) `ultimo::[a] -> (a -> Bool) -> N` que retorna el índice del último elemento de una lista para el cual se cumple un predicado dado, empezando desde 1. Si no hay ningún elemento que cumpla el predicado, debe devolverse 0.

19) `concat :: [[a]]-> [a]`, que concatena una lista de listas.
Por ejemplo: `concat [[0,1,2],[3],[4,5,6],[],[7,8]] = [0,1,2,3,4,5,6,7,8]`.

20) `lensum :: [[a]] -> N` que suma los largos de las listas de una lista de listas.
Por ejemplo: `lensum [[0,1,2],[3],[4,5,6],[],[7,8]] = 9`.

Ej.2. Pares Ordenados

El tipo de los pares ordenados se define en Haskell como sigue:

```
data (a,b) where { (_,_)::a -> b -> (a,b) }
```

(no es necesario escribir esta definición en su archivo, ya viene predefinida).

Defina las siguientes funciones:

1) `fst :: (a,b) -> a`, que devuelve el primer componente de un par.
   Por ejemplo: `fst(True,6) = True`.

2) `snd :: (a,b) -> b`, que devuelve el segundo componente de un par.
   Por ejemplo: `snd(True,(6,[1,2])) = (6,[1,2])`.

3) `zip :: [a]-> [b]-> [(a,b)]`, que recibe un par de listas y devuelve una lista que contiene los pares de elementos que aparecen en la misma posición en ambas listas. Si una lista es más larga que otra, no se consideran los elementos sobrantes.
   Por ejemplo: `zip [1,3,7,5] [True,False] = [(1,True),(3,False)]`.

4) `menMay :: Ord a => [a] -> a -> ([a],[a])`, que recibe una lista y un valor del mismo tipo que los elementos de la lista, y divide a la lista en dos: la primera conteniendo todos los elementos de la lista que son menores que el valor dado, y la segunda con todos los mayores o iguales.
   Por ejemplo: `menMay [3,6,8,2,1,5,9] 6 = ([3,2,1,5],[6,8,9])`.

Ej.3. Funciones parciales

Definimos la función `pre :: Bool -> String -> a -> a`, que permite definir una precondición a ser verificada *antes* de ejecutar el código de una función:

```
pre :: Bool -> String -> t -> t
pre = \b s p -> case b of { False -> error s  ;  True -> p }
```

Defina las siguientes funciones *parciales*, utilizando la función `pre` para especificar la precondición correspondiente en cada caso:

1) `head :: [a]-> a`, que devuelve el primer elemento de una lista no vacía.

2) `tail :: [a]-> [a]`, que devuelve el resultado de quitar el el primer elemento de una lista no vacía.

3) `last :: [a]-> a`, que devuelve el último elemento de una lista no vacía.

4) `init :: [a]-> [a]`, que devuelve el resultado de quitar el último elemento de una lista no vacía.

5) `(!!) :: [a]-> N-> a`, que devuelve el n-ésimo elemento de una lista, empezando desde `0`.

6) `minList :: Ord a =>[a]-> a`, que devuelve el mínimo elemento de una lista.

Ej.4. Más funciones sobre listas

Defina las siguientes funciones:

1) `prefijo :: Eq a => [a]-> [a]-> Bool`, que recibe dos listas y determina si la primera es prefijo de la segunda.
   Por ejemplo: `prefijo [1,2] [1,2,3,4,5] = True`.

2) `take :: N -> [a] -> [a]`, que recibe un natural `n` y una lista `l`, y devuelve la lista con los primeros `n` elementos de `l`. Si la lista tiene menos de `n` elementos, la devuelve entera. No usar funciones auxiliares.

3) `drop :: N -> [a] -> [a]`, que recibe un natural `n` y una lista `l`, y devuelve la lista resultante de quitar los primeros `n` elementos de `l`. Si la lista tiene menos de `n` elementos,devuelve la lista vacía. No usar funciones auxiliares.

4) `split :: N-> [a]-> ([a],[a])`, tal que `split n l` devuelve un par, conteniendo una lista con los primeros `n` elementos de `l` y otra lista con el resto de los elementos de `l`.
   Ejemplo: `partir 3 [2,4,6,8,9] = ([2,4,6],[8,9])`.

5) Dar otra definición de la función `split` de manera recursiva, sin utilizar funciones auxiliares.

6) `takeWhile :: (a->Bool)-> [a]-> [a]`, tal que `takeWhile p l` devuelve una lista formada con los primeros elementos de `l` para los cuales `p` se cumple (cuando `p` deja de cumplirse, no se tomarán más elementos de `l`).
   Ejemplo: `primerosQue even [0,2,3,4,6,7,8] = [0,2]` (`even` es un predicado definido en el Preludio de Haskell que determina si un número es par).

7) `dropWhile :: (a->Bool)-> [a]-> [a]`, tal que `dropWhile p xs` devuelve la lista resultante de quitar los primeros elementos de `xs` para los cuales `p` se cumple (cuando `p` deja de cumplirse, no se quitarán más elementos de `xs`, y se devolverá la lista resultante).
   Ejemplo: `dropWhile even [0,2,3,4,6,7,8] = [3,4,6,7,8]` .

8) `pares :: [a]->[a]`, que recibe una lista y se queda con aquellos elementos que están en las posiciones pares, contando el primer elemento como en posición 0.
   Por ejemplo: `pares [2,3,5,7,11,13,19] = [2,5,11,19]`.