FUNDAMENTOS DE LA COMPUTACIÓN
–LISTAS–
JUNIO 2017

El tipo (polimórfico) de las listas se define como:
`data [a] where { [] :: [a]  ;  (:) :: a -> [a] -> [a] }`

Para poder hacer estos ejercicios directamente en Haskell, necesitamos definir un módulo:
`module Listas where`

Como las funciones que vamos a definir ya existen en el Preludio de Haskell, debemos incluir la siguiente línea que evita que nuestras definiciones se superpongan con otras primitivas de Haskell:
`import Prelude hiding (null,length,sum,map,zip,zipWith,filter,and,or,any,all,(++), reverse,concat)`

Para simplificar el uso de algunas funciones que vamos a programar, usaremos el tipo de los enteros primitivo de Haskell llamado `Int`.
La expresión `case` para un entero no negativo `n` es la siguiente:
`case n of { 0 -> ...  ;  x+1 -> ... }`
Se deberá colocar la siguiente línea al principio del archivo para que esto se pueda utilizar:
`{-# LANGUAGE NPlusKPatterns #-}`

Ejercicios

Defina las siguientes funciones usando recursión estructural sobre listas:

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

2) `length::[a]-> Int`, 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::[Int]-> Int`, que suma todos los elementos de una lista de enteros.

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

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) `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 [1,3,5] [True,False] = [(1,True),(3,False)]`.

8) `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]`.

9) `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.

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

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

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

13) `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.

14) `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.

15) Redefina las tres funciones anteriores sin utilizar recursión directamente, sino funciones definidas anteriormente.

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

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

18) `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]`

19) `lensum::[[a]] -> Int` 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`