PROGRAMACIÓN FUNCIONAL 20 de Octubre de 2020

1. Definir una función `mapIf` que dada una función booleana `p`, una función unaria `f`, y una lista `xs`, produzca la lista que resulta de aplicar `f` a todos los elementos de `xs` que cumplen `p`, y mantener los elementos que no cumplen `p` tal como aparecen en `xs`. Por ejemplo

   ```
   mapIf odd (*2) [1,2,3,4,5,6,7,8]
   ⇒ [2,2,6,4,10,6,14,8]
   ```

   Se piden dos versiones: (a) recursiva usando los patrones de listas, y (b) usando listas intensionales.

2. Definir una función que calcule la lista más larga de una lista de listas. Si hay varias listas de la máxima longitud, cualquiera de ellas sirve como resultado. Se piden dos versiones: (a) recursiva usando los patrones de listas, y (b) usando `foldr1`.

3. El código binario reflejado o código Gray, nombrado así en honor del investigador Alessandre Frank Gray, es un sistema de numeración binario en el que dos valores sucesivos difieren solamente en uno de sus dígitos. El código Gray de n-bits es una lista de strings de n-bits construidos como sigue (en ese orden):

   ```
   n = 1: ["0","1"]
   n = 2: ["00","01","11","10"]
   n = 3: ["000","001","011","010","110","111","101","100"]
   n = 4: ["0000", "0001", "0011", "0010", "0110", "0111","0101",
           "0100", "1100","1101", "1111", "1110", "1010", "1011",
           "1001","1000"]
   etc.
   ```

   Los códigos de Gray para cada `n` se construyen de un modo determinado en función de los códigos para `n-1`. Por ejemplo, para n = 3,

   ```
   ["000","001","011","010",
           -- un 0 delante de los códigos de gray para n=2
   "110","111","101","100"]
          -- un 1 delante de los códigos de gray para n=2
          -- en orden inverso.
   ```

   Definir una función `gray:: Int -> [String]` que para cualquier `n`$\geq$`1` calcule la lista de strings de n-bits correspondiente. Por ejemplo:

   ```
   gray 3
   ⇒ ["000","001","011","010","110","111","101","100"]
   ```

   Indicaciones:

   - Usar `let` or `where` para que la llamada recursiva se calcule una sola vez.
   - La función `reverse :: [a] -> [a]` es predefinida.

4. La función `zipWith` es una generalización de la función `zip` que es *predefinida en Haskell* como sigue:

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
zipWith f (a:as) (b:bs) = f a b : zipWith f as bs
zipWith _ _ _ = []
```

Un ejemplo de uso es:

```
zipWith (+) [1,2,3] [3,5,9]
⇒ [4,7,11]
```

De hecho la función zip se define como

```
zip xs ys = zipWith pair xs ys
            where pair x y = (x, y)
```

o lo que es lo mismo `zip = zipWith (\x y -> (x,y))`.

Sea la siguiente función booleana

```
esCreciente :: [a] -> Bool
esCreciente [] = True
esCreciente  [x] = True
esCreciente (x:y:xs) = (x<y) && esCreciente (y:xs)
```

(a) Dar una definición alternativa de `esCreciente` que utize `zipWith` y `and`. Debe constar de una sóla ecuación.

(b) Generalizar la defición de la función `esCreciente` del apartado anterior para obtener una función

```
esOrdenada :: (a -> a -> Bool) -> [a] -> Bool
```

de modo que `esCreciente = esOrdenada (<)`.
Ejemplos de uso:

```
*Main> esOrdenada (<) [1,2,3]
True
*Main> esOrdenada (<=) [1,1,2,3]
True
*Main> esOrdenada (<) [1,1,2,3]
False
*Main> esOrdenada (>) [1,1,2,3]
False
*Main> esOrdenada (>=) [1,1,2,3]
False
```