Programación Funcional - Práctico 7

1. Para las siguientes funciones, determine su tipo y pase su implementación de notación **do** a uso de los operadores $(\gg=)$ y $(\gg)$:

   (a) $foo\ x\ y = \mathbf{do}\ a \leftarrow getLine$
   $\quad putStrLn\ (a +\!\!+ x)$
   $\quad b \leftarrow getLine$
   $\quad putStrLn\ (b +\!\!+ y)$
   $\quad putStrLn\ (a +\!\!+ b)$

   (b) $bar\ m1\ m2 = \mathbf{do}\ m1$
   $\quad x \leftarrow m2$
   $\quad y \leftarrow m1$
   $\quad return\ (x \vee y)$

   (c) $baz\ mf\ my\ x = \mathbf{do}\ f \leftarrow mf$
   $\quad y \leftarrow my$
   $\quad Just\ (f\ x\ y)$

2. La siguiente es una implementación de *quicksort* para listas sin elementos repetidos, donde el pivote elegido es siempre el primer elemento de la lista.

   $quicksort :: Ord\ a \Rightarrow [a] \to [a]$
   $quicksort\ [\,] = [\,]$
   $quicksort\ (x : xs) = \mathbf{let}\ smallers = quicksort\ [a \mid a \leftarrow xs, a < x]$
   $\quad biggers = quicksort\ [a \mid a \leftarrow xs, a > x]$
   $\quad \mathbf{in}\ smallers +\!\!+ [x] +\!\!+ biggers$

   Una variante no determinista de este algoritmo elegiría el pivote al azar. Implemente una versión *quicksortND*,

   $quicksortND :: Ord\ a \Rightarrow [a] \to [[a]]$

   que simule el comportamiento no determinístico utilizando la mónada *List* (de no determinismo):

   instance *Monad* [ ] **where**
   $return\ x = [x]$
   $xs \gg= f = [y \mid x \leftarrow xs, y \leftarrow f\ x]$

3. Dada la siguiente definición de árboles binarios con valores en los nodos:

   $$\textbf{data}\ BTree\ a = Empty \mid Node\ a\ (BTree\ a)\ (BTree\ a)$$

   Implemente la función:

   $$mapLevel :: BTree\ a \to Reader\ [a \to b]\ (BTree\ b)$$

   que dado un árbol, retorna una computación en la mónada *Reader* que dada una lista de funciones $[a \to b]$ retorna el árbol resultante de aplicar la primera función de la lista a los elementos del primer nivel del árbol original, la segunda función a los del segundo, etc. Asuma que la lista contiene al menos tantos elementos como niveles tiene el árbol.

   Por ejemplo, suponga que

   $$t = Node\ 8\ (Node\ 4\ Empty\ Empty)\ (Node\ 6\ (Node\ 10\ Empty\ Empty)\ Empty)$$

```
> runReader (mapLevel t) [(+1),(+2),(+3)]
Node 9 (Node 6 Empty Empty) (Node 8 (Node 13 Empty Empty) Empty)
```

4. Dada la siguiente definición de árboles binarios con valores en las hojas:

   $$LTree\ a = Leaf\ a \mid Join\ (LTree\ a)\ (LTree\ a)$$

   (a) Implemente la función:

   $$zipWithLT :: (a \to b \to c) \to LTree\ a \to State\ [b]\ (LTree\ c)$$

   que dada una función f y un árbol t, devuelve una computación en la mónada *State* (con una lista como estado), la cual retorna un árbol con la misma forma que t. Los elementos del nuevo árbol resultan de aplicar la función f a cada hoja de t y al valor que se encuentra en la cabeza de la lista (estado). En cada aplicación de f el valor en la cabeza de la lista (estado) es eliminado. La recorrida del árbol es en orden. Es decir, el comportamiento es muy similar al del *zipWith* estándar, pero ahora entre un árbol y una lista en lugar de entre dos listas. Asuma que la lista contiene al menos tantos elementos como el árbol.

   Por ejemplo, si $t = Join\ (Leaf\ 3)\ (Join\ (Leaf\ 2)\ (Leaf\ 10))$,

```
> fst $ runState (zipWithLT (+) t) [1,6,4]
Join (Leaf 4) (Join (Leaf 8) (Leaf 14))
```

   (b) Implemente la función:

   $$zipWithLT' :: (a \to b \to c) \to LTree\ a \to State\ ([b], b)\ (LTree\ c)$$

similar a la anterior, pero en la que el estado es un par $([b], b)$, de manera que si la cantidad de elementos de la lista es menor a la del árbol, para el resto de los elementos del árbol se utiliza siempre el segundo componente del par.

Por ejemplo, suponga que $t = \mathit{Join}\ (\mathit{Leaf}\ 3)\ (\mathit{Join}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 10))$:

```
> fst $ runState (zipWithLT' (+) t) ([4],1)
Join (Leaf 7) (Join (Leaf 3) (Leaf 11))
```

5. Se quiere tener una mónada *Cont* que implemente un contador global. Para esto se definen las funciones:

   - $\mathit{next} :: \mathit{Cont}\ \mathit{Int}$
     que retorna una computación *Cont* la cual retorna el valor corriente del contador y lo incrementa en uno.
   - $\mathit{runCont} :: \mathit{Cont}\ a \to a$
     que dada una computación *Cont* la ejecuta iniciando el contador en 0 y finalmente retorna su valor.

   Por ejemplo, si tenemos:

   $$\begin{aligned} m = \mathbf{do}\ & x \leftarrow \mathit{next} \\ & y \leftarrow \mathit{next} \\ & z \leftarrow \mathit{next} \\ & \mathit{return}\ [x, y, z] \end{aligned}$$

   Evaluar $(\mathit{runCont}\ m)$ retorna la lista $[0, 1, 2]$.

   Se pide definir el tipo $\mathit{Cont}\ a$, implementar la instancia de *Monad* para *Cont*, e implementar las funciones *next* y *runCont*.

6. Implemente la función:

   $$\mathit{inOrderCont} :: \mathit{LTree}\ a \to \mathit{Cont}\ [(\mathit{Int}, a)]$$

   que dado un árbol, retorna una computación en la mónada *Cont* que construye la lista resultante de una recorrida en orden del árbol, numerando los elementos a partir del 0.

   Por ejemplo, si $t = \mathit{Join}\ (\mathit{Leaf}\ \texttt{'w'})\ (\mathit{Join}\ (\mathit{Leaf}\ \texttt{'b'})\ (\mathit{Leaf}\ \texttt{'j'}))$:

```
> runCont (inOrderCont t)
[(0,'w'),(1,'b'),(2,'j')]
```