Programación Funcional - Práctico 4

1. Considere la siguiente definición de los números naturales:

   data $Nat = Zero \mid Succ\ Nat$

   Ejemplos de naturales son: $Zero$, $Succ\ Zero$, $Succ\ (Succ\ Zero)$, etc. De esta forma, la representación del n-ésimo natural es de la forma $Succ^n\ Zero$.

   (a) Defina las siguientes funciones sobre los naturales:

$nat2int :: Nat \to Int$	convierte un natural en entero
$duplica :: Nat \to Nat$	doble del argumento
$exp2 :: Nat \to Nat$	exponente en base 2
$suma :: Nat \to Nat \to Nat$	suma de naturales
$predecesor :: Nat \to Nat$	predecesor ($predecesor$ de cero es cero)

   (b) El fold para los naturales se define de la siguiente manera:

   $$\begin{aligned} &foldN :: (a \to a) \to a \to Nat \to a \\ &foldN\ h\ e\ Zero \quad\ = e \\ &foldN\ h\ e\ (Succ\ n) = h\ (foldN\ h\ e\ n) \end{aligned}$$

   Defina las funciones de la parte a) en función de $foldN$.

   (c) Defina la función $fib :: Nat \to Nat$ que computa los números de fibonacci. Una forma de definir esta función es por la conocida fórmula de recurrencia que caracteriza estos números. Impleméntela.

   Es bien sabido que dicha definición tiene un problema: su comportamiento es exponencial. Pruebe aplicarla con valores crecientes de naturales y lo podrá verificar. Hay una mejor definición, que es lineal, la cual utiliza una forma "iterativa" para computar los números de fibonacci. Implemente esta definición alternativa.

2. Suponga que definimos los enteros de la siguinte forma:

   data $OurInt = IntZero \mid Pos\ Nat \mid Neg\ Nat$

   tal que $IntZero$ representa el cero de los enteros, Pos los enteros positivos (1,2,...) y Neg los negativos (-1,-2,...). Por ejemplo, el 2 es dado por $Pos\ (Succ\ Zero)$, mientras que el -1 por $Neg\ Zero$.

(a) Defina la instancia de la clase *Num* para *OurInt*.

(b) ¿Qué problema tiene esta otra representación?

$$\textbf{data}\ \mathit{OtroInt} = \mathit{OZero} \mid \mathit{OPos}\ \mathit{OtroInt} \mid \mathit{ONeg}\ \mathit{OtroInt}$$

O dicho de otra forma, que propiedad tiene la definición de *OurInt* que no la tiene *OtroInt*?

¿Y esta otra?

$$\textbf{data}\ \mathit{OtroInt}' = \mathit{OPos}\ \mathit{Nat} \mid \mathit{ONeg}\ \mathit{Nat}$$

3. Considere la siguiente definición de árbol binario:

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

(a) Defina las recorridas en inorder, preorder y postorder sobre un árbol binario, las cuales listan los elementos del árbol en el respectivo orden. Todas ellas tienen tipo $\mathit{Tree}\ a \to [a]$.

(b) Defina la función $\mathit{mkTree} :: \mathit{Ord}\ a \Rightarrow [a] \to \mathit{Tree}\ a$ que construye un árbol binario de búsqueda a partir de una lista. (El árbol generado no precisa estar balanceado.)

(c) Que hace la composición $\mathit{inorder} \circ \mathit{mkTree}$?

4. Considere el tipo de los árboles binarios con información en las hojas:

$$\textbf{data}\ \mathit{BTree}\ a = \mathit{Leaf}\ a \mid \mathit{Fork}\ (\mathit{BTree}\ a)\ (\mathit{BTree}\ a)$$

(a) Defina la función $\mathit{depths} :: \mathit{BTree}\ a \to \mathit{BTree}\ \mathit{Int}$ que reemplaza el valor en cada hoja del árbol por su profundidad en el mismo. La profundidad de la raíz es cero. Por ejemplo:

$$\begin{array}{l}\mathit{depths}\ (\mathit{Fork}\ (\mathit{Fork}\ (\mathit{Leaf}\ \texttt{'a'})\ (\mathit{Leaf}\ \texttt{'b'}))\ (\mathit{Leaf}\ \texttt{'c'})) \\ = \\ \mathit{Fork}\ (\mathit{Fork}\ (\mathit{Leaf}\ 2)\ (\mathit{Leaf}\ 2))\ (\mathit{Leaf}\ 1)\end{array}$$

(b) Defina la función $\mathit{balanced} :: \mathit{BTree}\ a \to \mathit{Bool}$ que determina si un árbol está balanceado. Entendemos por balanceado si el número de hojas en los subárboles izquierdo y derecho de todo nodo difiere a lo más en uno. Las hojas se consideran balanceadas. Sugerencia: primero defina una función *size* que compute el número de hojas de un árbol.

(c) Defina la función $\mathit{mkBTree} :: [a] \to \mathit{BTree}\ a$ que convierte una lista no vacía de valores de tipo a en un árbol balanceado. Sugerencia: primero defina una función que parta una lista en dos mitades cuyo largo difiera a lo más en uno.

(d) Defina la función $retrieve :: BTree\ a \to Int \to a$ que retorna el valor contenido en la n-ésima hoja de un árbol contada de izquierda a derecha. El valor n es pasado como uno de los parámetros. Las hojas se empiezan a numerar desde 1.

5. El tipo de los árboles binarios homogéneos:

$$\textbf{data}\ HTree\ a = Tip\ a \mid Bin\ (HTree\ a)\ a\ (HTree\ a)$$

representa árboles binarios que contienen información tanto en sus nodos como en sus hojas. Dichos árboles están emparentados tanto con los árboles de tipo $Tree\ a$ como con los de tipo $BTree\ a$.

(a) Defina la correspondiente función map para este tipo:

$$mapHT :: (a \to b) \to (HTree\ a \to HTree\ b)$$

(b) Defina una función $subtrees :: BTree\ a \to HTree\ (BTree\ a)$ que dado un árbol binario con información en las hojas computa un árbol homogéneo con su misma forma y tal que el valor almacenado en cada nodo contiene el correspondiente árbol binario que tiene raíz en ese nodo en el árbol original. Por ejemplo:

$$\begin{array}{l} subtrees\ (Fork\ (Leaf\ 2)\ (Fork\ (Leaf\ 3)\ (Leaf\ 4))) \\ = \\ Bin\ (Tip\ \ (Leaf\ 2)) \\ \quad (Fork\ (Leaf\ 2)\ (Fork\ (Leaf\ 3)\ (Leaf\ 4))) \\ \quad (Bin\ \ (Tip\ (Leaf\ 3)) \\ \qquad\quad (Fork\ (Leaf\ 3)\ (Leaf\ 4)) \\ \qquad\quad (Tip\ (Leaf\ 4))) \end{array}$$

(c) Defina la función $sizes :: BTree\ a \to HTree\ Int$ que dado un árbol binario retorna el árbol homogéneo que en cada nodo contiene el número de hojas que tiene el correspondiente árbol binario con raíz en ese nodo. Por ejemplo:

$$\begin{array}{l} sizes\ (Fork\ (Leaf\ 2)\ (Fork\ (Leaf\ 3)\ (Leaf\ 4))) \\ = \\ Bin\ (Tip\ 1)\ 3\ (Bin\ (Tip\ 1)\ 2\ (Tip\ 1)) \end{array}$$

(d) Defina la función sizes ahora usando las partes a) y b).

6. Se requiere implementar una función sobrecargada *size* que computa una noción de tamaño.

(a) Declare una clase "$Sizeable\ a$" con un método $size :: a \to Int$. Declare instancias de *Sizeable* para los tipos *Int* y *Char*. El tamaño de un entero viene dado por su valor absoluto. El tamaño de un carácter es 1.

(b) Declare instancias de *Sizeable* para listas y pares. El tamaño de una lista es la suma de los tamaños de los valores que contiene. Por ejemplo *size* $[1, 2, -3] \equiv 6$. El tamaño de los pares es la suma de los tamaños de sus componentes, por ejemplo *size* $(\texttt{'a'}, [\,]) \equiv 1$

(c) Declare una instancia de *Sizeable* para el tipo *Tree* definido en el ejercicio 3. El tamaño del árbol viene dado por la suma de los tamaños de los valores que contiene.

(d) Implemente una función *filtLt* que dado una lista de valores de un tipo instancia de *Sizeable* y un entero n, retorna solamente aquellos que tienen tamaño menor que n.

(e) Implemente una función *isSmaller* que dados dos valores de tipos *Sizeable*s (pueden ser tipos distintos), computa un booleano indicando si el primero es más pequeño que el segundo.

(f) Declare una clase "*Enumerate a*" como subclase de "*Sizeable a*" con un método $enum : Int \to [a]$. La función *enum*, dado un entero n, retorna todos los valores del tipo a con tamaño menor o igual a n. Declare instancias de *Enumerate* para *Int*, *Char* y pares.