Práctica 3

1. El modelo de color RGB es un modelo aditivo que tiene al rojo, verde y azul como colores primarios. Cualquier otro color se expresa en términos de los porcentajes de cada uno estos tres colores que es necesario combinar en forma aditiva para obtenerlo. Dichas proporciones caracterizan a cada color de manera biunívoca, por lo que usualmente se utilizan estos valores como representación de un color.
Definir un tipo *Color* en este modelo y una función *mezclar* que permita obtener el promedio componente a componente entre dos colores.

2. Consideremos un editor de líneas simple. Supongamos que una *Línea* es una secuencia de caracteres $c_1, c_2, \ldots, c_n$ junto con una posición p, siendo $0 \leqslant p \leqslant n$, llamada cursor (consideraremos al cursor a la derecha de un caracter que será borrado o insertado, es decir como el cursor de la mayoría de los editores). Se requieren las siguientes operaciones sobre líneas:

$$
\begin{array}{ll}
\mathit{vacía} & :: \mathit{Línea} \\
\mathit{moverIzq} & :: \mathit{Línea} \to \mathit{Línea} \\
\mathit{moverDer} & :: \mathit{Línea} \to \mathit{Línea} \\
\mathit{moverIni} & :: \mathit{Línea} \to \mathit{Línea} \\
\mathit{moverFin} & :: \mathit{Línea} \to \mathit{Línea} \\
\mathit{insertar} & :: \mathit{Char} \to \mathit{Línea} \to \mathit{Línea} \\
\mathit{borrar} & :: \mathit{Línea} \to \mathit{Línea}
\end{array}
$$

La descripción informal es la siguiente: (1) la constante *vacía* denota la línea vacía, (2) la operación *moverIzq* mueve el cursor una posición a la izquierda (siempre que ello sea posible), (3) análogamente para *moverDer*, (4) *moverIni* mueve el cursor al comienzo de la línea, (5) *moverFin* mueve el cursor al final de la línea, (6) la operación *borrar* elimina el caracterer que se encuentra a la izquierda del cursor, (7) *insertar* agrega un caracter en el lugar donde se encontraba el cursor, dejando al caracter insertado a su izquierda.
Definir un tipo de datos *Línea* e implementar las operaciones dadas.

3. Dado el tipo de datos

data *CList a* = *EmptyCL* | *CUnit a* | *Consnoc a* (*CList a*) *a*

a) Implementar las operaciones de este tipo algebraico teniendo en cuenta que:

- Las funciones de acceso son *headCL*, *tailCL*, *isEmptyCL*, *isCUnit*.
- *headCL* y *tailCL* no están definidos para una lista vacía.
- *headCL* toma una *CList* y devuelve el primer elemento de la misma (el de más a la izquierda).
- *tailCL* toma una *CList* y devuelve la misma sin el primer elemento.
- *isEmptyCL* aplicado a una *CList* devuelve *True* si la *CList* es vacía (*EmptyCL*) o *False* en caso contrario.
- *isCUnit* aplicado a una *CList* devuelve *True* sii la *CList* tiene un solo elemento (*CUnit a*) o *False* en caso contrario.

b) Definir una función *reverseCL* que toma una *CList* y devuelve su inversa.

c) Definir una función *inits* que toma una *CList* y devuelve una *CList* con todos los posibles inicios de la *CList*.

d) Definir una función *lasts* que toma una *CList* y devuelve una *CList* con todas las posibles terminaciones de la *CList*.

e) Definir una función *concatCL* que toma una *CList* de *CList* y devuelve la *CList* con todas ellas concatenadas

4. Dado el siguiente tipo algebraico:

data *Aexp* = *Num Int* | *Prod Aexp Aexp* | *Div Aexp Aexp*

a) Defina un evaluador $eval :: Aexp \to Int$. ¿Cómo maneja los errores de división por 0?

b) Defina un evaluador $seval :: Aexp \to Maybe\ Int$.

5. Si un árbol binario es dado como un nodo con dos subárboles idénticos se puede aplicar la técnica *sharing* para que los subárboles sean representados por el mismo árbol. Definir las siguientes funciones de manera que se puedan compartir la mayor cantidad posible de elementos de los árboles creados:

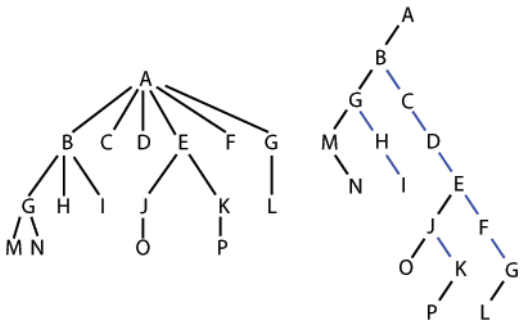
a) $completo :: a \to Int \to Tree\ a$, tal que dado un valor x de tipo a y un entero d, crea un árbol binario completo de altura d con el valor x en cada nodo.

b) $balanceado :: a \to Int \to Tree\ a$, tal que dado un valor x de tipo a y un entero n, crea un árbol binario balanceado de tamaño n, con el valor x en cada nodo.

6. Dada las siguientes representaciones de árboles generales y de árboles binarios

data *GenTree a* = *EmptyG* | *NodeG a* [*GenTree a*]
data *BinTree a* = *EmptyB* | *NodeB* (*BinTree a*) *a* (*BinTree a*)

defina una función *g2bt* que dado un árbol nos devuelva un árbol binario de la siguiente manera:



La función *g2bt* reemplaza cada nodo n del árbol general (*NodeG*) por un nodo n' del árbol binario (*NodeB*), donde el hijo izquierdo de n' representa el hijo más izquierdo de n, y el hijo derecho de n' representa al hermano derecho de n, si existiese (observar que de esta forma, el hijo derecho de la raíz es siempre vacío).

7. Definir las siguientes funciones sobre árboles binarios de búsqueda (bst):

1. $maximum :: Ord\ a \Rightarrow BST\ a \to a$, que calcula el máximo valor en un bst.
2. $checkBST :: Ord\ a \Rightarrow BST\ a \to Bool$, que chequea si un árbol binario es un bst.

8. La definición de *member* dada en teoría (la cual determina si un elemento está en un bst) realiza en el peor caso $2 * d$ comparaciones, donde d es la altura del árbol. Dar una definición de *member* que realice a lo sumo $d + 1$ comparaciones. Para ello definir *member* en términos de una función auxiliar que tenga como parámetro el elemento candidato, el cual puede ser igual al elemento que se desea buscar (por ejemplo, el último elemento para el cual la comparación de $a \leqslant b$ retornó True) y que chequee que los elementos son iguales sólo cuando llega a una hoja del árbol.

9. La función *insert* dada en teoría para insertar un elemento en un rbt puede optimizarse elimando comparaciones innecesarias hechas por la función *balance*. Por ejemplo, en la definición de la función *ins* cuando se aplica *balance* sobre el resultado de aplicar *insert x* sobre el subárbol izquierdo (l) y el subárbol derecho (r), los casos de *balance* para testear que se viola el invariante 1 en el subárbol derecho no son necesarios dado que r es un rbt.

a) Definir dos funciones *lbalance* y *rbalance* que chequeen que el invariante 1 se cumple en los subárboles izquierdo y derecho respectivamente.

b) Reemplazar las llamadas a *balance* en *ins* por llamadas a alguna de estas dos funciones.

10. Definir una función $fromList :: [a] \to Heap\ a$, que cree un leftist heap a partir de una lista, convirtiendo cada elemento de la lista en un heap de un solo elemento y aplicando la función *merge* hasta obtener un solo heap. Aplicar la función *merge* $\lceil \lg n \rceil$ veces, donde n es la longitud de la lista que recibe como argumento la función, de manera que *fromList* sea de orden $O(n)$.