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

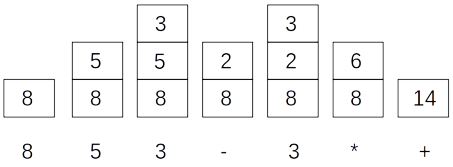
4. Defina un evaluador $eval :: Exp \to Int$ para el siguiente tipo algebraico:

data $Exp = Lit\ Int \mid Add\ Exp\ Exp \mid Sub\ Exp\ Exp \mid Prod\ Exp\ Exp \mid Div\ Exp\ Exp$

5. La *notación polaca inversa* o *RPN* (del inglés *Reverse Polish Notation*) es una manera alternativa de escribir expresiones matemáticas, en la cual los operadores se escriben luego de los operandos, es decir, usando notación *posfija*. Por ejemplo, la suma de los enteros 3 y 5 que con notación *infija* notamos $3 + 5$, en RPN se escribe $3\ 5\ +$. La siguiente tabla muestra más ejemplos de expresiones aritméticas, escritas con notación infija y posfija:

Notación infija	Notación polaca inversa
$(2 + 3) * 5$	$2\ 3\ +\ 5\ *$
$15/(9 - 4)$	$15\ 9\ 4\ -\ /$
$(3 - 1) * (2 + 4)$	$3\ 1\ -\ 2\ 4\ +\ *$
$8 + (5 - 3) * 3$	$8\ 5\ 3\ -\ 3\ *\ +$

Para evaluar una expresión escrita en RPN, podemos usar un *stack* o *pila*. Recorremos la expresión de izquierda a derecha. Cada vez que encontramos un número, lo apilamos. Cada vez que encontramos un operador, retiramos los dos números que están en la cima de la pila, le aplicamos el operador y apilamos el resultado. Si la expresión está bien formada, al alcanzar el final de la misma, debemos tener un único número en la pila, que representa el resultado de la expresión.
Por ejemplo, para la expresión $8\ 5\ 3\ -\ 3\ *\ +$, el algoritmo anterior realiza los siguientes pasos:



Una de las ventajas de esta notación es que elimina la necesidad del uso de paréntesis y de reglas de precedencia de operadores, ya que el proceso de apilamiento determina el orden en que deben computarse las operaciones.

a) Defina una función $parseRPN :: String \to Exp$ que, dado un string que representa una expresión escrita en RPN, construya un elemento del tipo *Exp* presentado en el ejercicio 4 correspondiente a la expresión dada. Por ejemplo:

$parseRPN$ "$8\ 5\ 3 - 3 * +$" $= Add\ (Lit\ 8)\ (Prod\ (Sub\ (Lit\ 5)\ (Lit\ 3))\ (Lit\ 3))$

Ayuda: para implementar *parseRPN* puede seguir un algoritmo similar al presentado anteriormente. En lugar de evaluar las expresiones, debe construir un valor de tipo *Exp*.

b) Defina una función $evalRPN :: String \to Int$ para evaluar expresiones aritméticas escritas en RPN. Por ejemplo:

$evalRPN$ "$8\ 5\ 3 - 3 * +$" $= 14$

Ayuda: use las funciones *parseRPN* y *eval* definidas anteriormente.

6.

a) Considere el evaluador $eval :: Exp \to Int$ del ejercicio 4. ¿Cómo maneja los errores de división por 0?

b) Defina un evaluador $seval :: Exp \to Maybe\ Int$ para controlar los errores de división por 0.