Programación Funcional

Aclaraciones:

- *Esta evaluación es a libro abierto. Se pueden usar todas las funciones y propiedades vistas en clase, aclarando la referencia. Cualquier otra función o propiedad que se utilice* ***debe ser definida o demostrada****.*
- *No se olvide de poner nombre, nro. de alumno, nro. de hoja y cantidad total de hojas en cada una de las hojas.*
- *Le recomendamos leer el enunciado en su totalidad y organizar sus ideas antes de comenzar la resolución.*
- *Recuerde que reusar código es una forma muy eficiente de disminuir el tiempo necesario para programar.*
- *La intención de la evaluación es medir cuánto comprende usted del tema. Por ello, no dude en escribir todo lo que sabe, explicar lo que se propone antes de escribir código y probar sus funciones con ejemplos.*

Un árbol ternario de búsqueda de prefijos es una estructura que permite operar eficientemente con cadenas de elementos y sus prefijos (donde un prefijo es cualquier subcadena comprendida desde el principio de la original y de longitud menor o igual). Considere la siguiente representación para estos árboles:

```
data (Ord a) => PrefixT a =
                Nil |         -- un árbol vacío
                Node a        -- un árbol con un prefijo a
                  (PrefixT a) -- un subárbol con prefijos menores que a
                  (PrefixT a) -- un subárbol con prefijos cuyo primer elemento es a
                  (PrefixT a) -- un subárbol con prefijos mayores que a
```

En resumen los árboles ternarios de búsqueda de prefijos mantienen un invariante similar al de los árboles de búsqueda, en el sentido que para un nodo dado su hijo izquierdo contiene los prefijos cuyo primer elemento es menor que el del nodo, mientras que el hijo derecho contiene los prefijos cuyo primer elemento es mayor. Por otro lado el hijo central es el árbol de búsqueda de prefijos que contiene todos los prefijos que tienen como primer elemento el del nodo padre.

Ejercicio 1
Escriba las siguientes funciones:

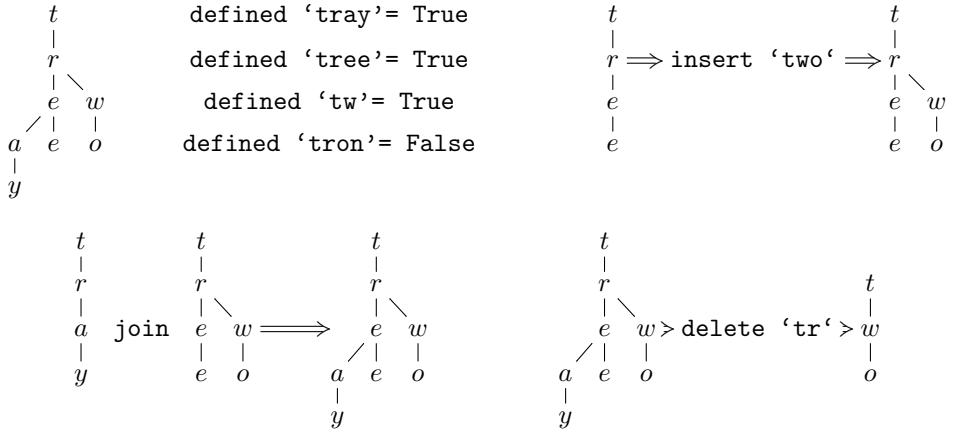
a) `defined :: Ord a => PrefixT a -> [a] -> Bool`
Que retorna verdadero si la cadena dada es un prefijo definido en el árbol (note que la cadena vacía es prefijo de cualquier cadena).

b) `insert :: Ord a => PrefixT a -> [a] -> PrefixT a`
Que inserta una nueva cadena al árbol (definiendo todos sus prefijos).

c) `join :: Ord a => PrefixT a -> PrefixT a -> PrefixT a`
Que une dos árboles de prefijos, es decir, el árbol resultado contiene todos los prefijos definidos en los argumentos.

d) `delete :: Ord a => PrefixT a -> [a] -> PrefixT a`
Que elimina del árbol de prefijos todos los elementos con el prefijo dado.

Ejemplos:



Ejercicio 2
Dar el tipo y escribir una función `foldPrefixT` que generalice la recursión sobre estructuras `PrefixT`. Luego escriba las siguientes funciones sin utilizar recursión explícita (i.e. utilizando esquemas de recursión), puede utilizar las funciones del punto 1 sin reescribirlas. Tenga en cuenta que no en todas es necesario usar `foldPrefixT` y que puede usar todas las funciones vistas en clase.

a) `flatten :: Ord a => PrefixT a -> [[a]]`
Que retorna todos los prefijos en el árbol en orden lexicográfico y sin repetidos.

b) `insertAll :: Ord a => [[a]] -> PrefixT a`
Que dada una lista de cadenas construye el árbol con todas las cadenas y sus prefijos definidos.

c) `union :: Ord a => [PrefixT a] -> PrefixT a`
Que dada una lista de árboles retorna un árbol que tiene definido los prefijos de cada árbol en la secuencia.

d) `intersect :: Ord a => PrefixT a -> PrefixT a -> PrefixT a`
Que dados dos árboles rertorna un árbol con los prefijos que están en ambos árboles.

e) `palindromes :: Ord a => PrefixT a -> PrefixT a`
Que construye un árbol con todos los prefijos que son un palindromo de algún prefijo definido en el árbol de entrada (una cadena es un palindromo de otra cuando al invertirla resulta igual, por ejemplo "neuquen" y "anana" (sin considerar las tildes).

f) `trim :: Ord a => [[a]] -> [[a]]`
Que dada una lista de cadenas retorna una de longitud menor o igual donde las cadenas prefijas de otras han sido eliminadas. Por ejemplo, ["prefijo", "pre", "cadena", "cadena"] debería retornar ["prefijo", "cadena"], pues "pre" es prefijo de "prefijo" y "cadena" es prefijo de "cadena".

g) `longest :: Ord a => PrefixT a -> Int`
Que retorna la longitud del camino más largo entre dos nodos cualesquiera.

Ejercicio 3
Considere las siguientes formas de árboles binarios:

```
data AB a b    = Leaf b | NodeAB a (AB a b) (AB a b)
data Tree a    = EmptyT | NodeT a (Tree a) (Tree a)
data TipTree a = Tip a  | Join (TipTree a) (TipTree a)
```

Considere, además, las siguientes funciones:

```
skeleton :: AB a b -> Tree a
skeleton (Leaf b)       = EmptyT
skeleton (NodeAB a l r) = NodeT a (skeleton l) (skeleton r)

extract :: AB a b -> TipTree b
extract (Leaf b)       = Tip b
extract (NodeAB a l r) = Join (extract l) (extract r)

couple :: Tree a -> TipTree b -> AB a b
couple EmptyT (Tip x) = Leaf x
couple (NodeT x t1 t2) (Join tt1 tt2) = NodeAB x (couple t1 tt1) (couple t2 tt2)

decouple :: AB a b -> (Tree a, TipTree b)
decouple ab = (skeleton ab, extract ab)
```

a) Demustre que: `uncurry couple . decouple = id`