Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3661 - Taller de Lenguajes de Programación I
Enero-Marzo 2011

Quiz Haskell

(10 puntos)

Nombre:

Carnet:

Atención: en todas sus respuestas debe utilizar *exclusivamente* funciones disponibles en el Preludio Haskell. Si Ud. no sabe si una función está o no en el Preludio Haskell, puede preguntarlo *públicamente* para recibir un **si** o **no** como respuesta y **nada** más.

1. Considere el siguiente tipo de datos recursivo para representar Arboles Rose Etiquetados

```
data Rose a b = Empty | Leaf b | Node a [Rose a b]
```

que permite escribir árboles de la forma

```
Node '*' [ Leaf 42, Empty,
           Node 'a' [Leaf 97, Empty, Node 'b' []],
           Empty, Node 'c' [] ]
```

En este ejemplo está parametrizado como `Rose Char Int`, pero el tipo es un contenedor *general*.

a) **(3 puntos)** Escriba la función `foldR` que realiza el *fold* genérico sobre un valor del tipo de datos `Rose a b` para transformarlo en un valor del tipo `c`, indicando la firma más general posible.

Para construir un *fold* genérico es necesario proveer a la función *fold* concreta una función parámetro por cada constructor del tipo a procesar. Cada función consolida el valor contenido en el constructor particular, en un valor único resultante del *fold*. En nuestro caso, el tipo tiene tres constructores, uno para el árbol vacío, otro para las hojas y áquel para los nodos internos, de manera que hará falta un valor base para el constructor de árboles vacíos además de dos funciones, una para procesar los contenidos de las hojas produciendo un valor único, y otra para procesar los contenidos de los nodos internos que como tienen un árbol dentro tienen que traer valores consolidados previamente, así

```
e  :: c               -- Valor base
fl :: b -> c          -- Función para hojas
fn :: a -> [c] -> c   -- Función para nodos
```

entonces, podemos construir el fold genérico para este tipo de datos

```
foldR :: (a -> [c] -> c) -> (b -> c) -> c -> Rose a b -> c
foldR fn fl e Empty       = e
foldR fn fl e (Leaf b)    = fl b
foldR fn fl e (Node a rs) = fn a $ map (foldR fn fl e) rs
```

que refleja la estructura recursiva de un *fold*: emitir el valor base en los árboles vacíos, procesar los contenidos de las hojas con la función que los transforma, y para el caso de los nodos, que tienen presencia recursiva del tipo `Rose a b`, primero aplicando el `foldR` sobre los árboles contenidos en la lista, y luego aplicando la función que transforma los contenidos de la lista resultante.

b) **(2 puntos)** Utilice la función `foldR` para escribir la función de orden superior `mapR` que realiza la transformación genérica de contenidos sobre un valor del tipo de datos `Rose a b` preservando su estructura, indicando la firma más general posible.

Un *map* genérico opera como un *fold* que preserva la estructura. Esto es, después de aplicar las funciones de transformación necesaria en cada elemento de la estrutura, debe reconstruirse la estructura con los valores resultantes. Por lo tanto, necesitaremos una función por cada constructor **transformable** en la estructura, que en nuestro caso se reduce a dos funciones, una para transformar los contenidos de las hojas y otra para transformar los contenidos de un nodo, así

```
tl :: b -> d   -- Para transformar hojas
tn :: a -> c   -- Para transformar nodos
```

Los árboles vacíos permanecerán vacíos, por lo que esté será el valor base para el uso de `foldR`, así que disponemos de todos los elementos necesarios para escribir `mapR` como

```
mapR :: (a -> c) -> (b -> d) -> Rose a b -> Rose c d
mapR tn tl = foldR fn fl Empty
             where fl b    = Leaf $ tl b
                   fn a rs = Node (tn a) rs
```

c) **(2 puntos)** Utilice la función `foldR` o la función `mapR` según convenga para escribir la función `leaves`, que retorna una tupla de listas conteniendo, respectivamente, todos los valores almacenados en las hojas y los nodos del Arbol Rose Etiquetado, siguiendo el recorrido en orden previo (*preorder* procesando los hijos de izquierda a derecha).

Para construir la función `leaves` solicitada usando `foldR`, es necesario determinar cuáles funciones son necesarias para los casos árbol vacío, hoja y nodo. En el caso de un árbol vacío, hay que producir una tupla de listas vacías (después de todo, no hay hijos que listar); en el caso de laa hoja, simplemente hay que retornar una tupla con la primera lista vacía y cuya segunda lista contenga únicamente el valor almacenado en la hoja. En el caso de un nodo, hay que recuperar todas las posibles listas que vengan en el segundo elemento de las tuplas, concatenarlas e incorporar el elemento contenido en el nodo como primer elemento de la primera lista para así producir en preorden, por lo tanto

```
leaves :: Rose a b -> ([a],[b])
leaves = foldR fn fl ([],[])
           where fl b    = ([],[b])
                 fn a rs = (a:concat as, concat bs)
                                where (as,bs) = unzip rs
```

2. **(3 puntos)** Definimos un *alfabeto* como un conjunto de símbolos cualesquiera, modelados con el tipo de datos

```
type Alphabet = [Char]
```

Una *palabra* es cualquier secuencia finita de símbolos tomados del alfabeto; note que la palabra que no tiene símbolo alguno es la palabra *vacía*. Definimos como Σ^* al conjunto infinito de todas las palabras que pueden construirse a partir de un alfabeto, incluyendo la palabra vacía. Escriba la función

```
sigma :: Alphabet -> [[Char]]
```

que enumere las palabras del alfabeto en orden según su longitud y sin repetir ninguna palabra, e.g.

```
ghci> take 10 $ sigma "ab"
["","a","b","aa","ba","ab","bb","aaa","baa","aba"]
```

Si una palabra $w \in \Sigma^*$ entonces $\forall a \in \Sigma \Rightarrow aw \in \Sigma^*$. La palabra vacía es la de menor longitud, así que debe aparecer primero en la lista, el resto de la lista se construye aplicando la regla anterior sobre cada palabra de la lista. Esto se expresa de manera muy sucinta con una lista por comprensión

```
sigma as = if null as then []
                      else [] : [ a : w | w <- sigma as, a <- as ]
```