Trabajo Práctico # 1 bis

Programación Funcional, Universidad Nacional de Quilmes

24 de marzo de 2018

Aclaraciones:

- *Los ejercicios fueron pensados para ser resueltos en el orden en que son presentados. No se saltee ejercicios sin consultar antes a un docente.*
- *Recuerde que puede aprovechar en todo momento las funciones que ha definido, tanto las de esta misma práctica como las de prácticas anteriores.*
- *Pruebe todas sus implementaciones, al menos en una consola interactiva.*
- *Es sumamente aconsejable resolver los ejercicios utilizando primordialmente los conceptos y metodologías vistos en clase, dado que los exámenes de la materia evaluarán principalmente este aspecto. Si se encuentra utilizando formas alternativas al resolver los ejercicios consulte a los docentes.*
- *No dude en manifestar observaciones y críticas sobre los ejercicios de esta práctica, que con gusto serán recibidas por los docentes.*
- *Los ejercicios del anexo pueden obviarse, pero recuerde que aportan una compresión más profunda sobre los temas que aborda esta práctica. Considere resolverlos si se encuentra practicando para una instancia de evaluación y ya resolvió todos los anteriores.*

1. Árboles binarios

Defina las siguientes funciones sobre árboles binarios utilizando recursión estructural según corresponda:

1. `sumT :: Tree Int -> Int`
   Dado un árbol binario de enteros devuelve la suma entre sus elementos.

2. `sizeT :: Tree a -> Int`
   Dado un árbol binario devuelve su cantidad de elementos, es decir, el tamaño del árbol (size en inglés).

3. `mapDoubleT :: Tree Int -> Tree Int`
   Dado un árbol de enteros devuelve un árbol con el doble de cada número.

4. `mapLengthT :: Tree String -> Tree Int`
   Dado un árbol de palabras devuelve un árbol con la longitud de cada palabra.

5. `elemT :: Eq a => a -> Tree a -> Bool`
   Dados un elemento y un árbol binario devuelve True si existe un elemento igual a ese en el árbol.

6. `occurrsT :: Eq a => a -> Tree a -> Int`
   Dados un elemento e y un árbol binario devuelve la cantidad de elementos del árbol que son iguales a e.

7. `averageT :: Tree Persona -> Int`
   Dado un árbol de personas devuelve el promedio entre las edades de todas las personas. Definir las subtareas que sean necesarias para resolver esta función.
   Nota: Utilizar el tipo Persona ya definido.

8. `countLeaves :: Tree a -> Int`
   Dado un árbol devuelve su cantidad de hojas.
   Nota: una hoja (leaf en inglés) es un nodo que no tiene hijos.

9. `leaves :: Tree a -> [a]`
   Dado un árbol devuelve los elementos que se encuentran en sus hojas.

10. `heightT :: Tree a -> Int`
    Dado un árbol devuelve su altura.
    Nota: la altura (height en inglés) de un árbol es la cantidad máxima de nodos entre la raíz y alguna de sus hojas. La altura de un árbol vacío es cero y la de una hoja es 1.

11. `countNoLeaves :: Tree a -> Int`
    Dado un árbol devuelve el número de nodos que no son hojas. ¿Cómo podría resolverla sin utilizar recursión? Primero defínala con recursión y después sin ella.

12. `mirrorT :: Tree a -> Tree a`
    Dado un árbol devuelve el árbol resultante de intercambiar el hijo izquierdo con el derecho, en cada nodo del árbol.

13. `listInOrder :: Tree a -> [a]`
    Dado un árbol devuelve una lista que representa el resultado de recorrerlo en modo *in-order*.
    Nota: En el modo in-order primero se procesan los elementos del hijo izquierdo, luego la raiz y luego los elementos del hijo derecho.

14. `listPreOrder :: Tree a -> [a]`
    Dado un árbol devuelve una lista que representa el resultado de recorrerlo en modo *pre-order*.
    Nota: En el modo pre-order primero se procesa la raiz, luego los elementos del hijo izquierdo, a continuación los elementos del hijo derecho.

15. `listPosOrder :: Tree a -> [a]`
    Dado un árbol devuelve una lista que representa el resultado de recorrerlo en modo *post-order*.
    Nota: En el modo post-order primero se procesan los elementos del hijo izquierdo, a continuación los elementos del hijo derecho y finalmente la raiz.

16. `concatT :: Tree [a] -> [a]`
    Dado un árbol de listas devuelve la concatenación de todas esas listas. El recorrido debe ser in-order.

17. `levelN :: Int -> Tree a -> [a]`
    Dados un número n y un árbol devuelve una lista con los nodos de nivel n.
    Nota: El primer nivel de un árbol (su raíz) es 0.

18. `listPerLevel :: Tree a -> [[a]]`
    Dado un árbol devuelve una lista de listas en la que cada elemento representa un nivel de dicho árbol.

19. `widthT :: Tree a -> Int`
    Dado un árbol devuelve su ancho (width en inglés), que es la cantidad de nodos del nivel con mayor cantidad de nodos.

20. `leftBranches :: Tree a -> [a]`
    Devuelve todos los elementos encontrados en el camino de todas las ramas derechas.

21. `longestBranch :: Tree a -> [a]`
    Devuelve los elementos de la rama más larga del árbol

22. `allPaths :: Tree a -> [[a]]`
    Dado un árbol devuelve todos los caminos, es decir, los caminos desde la raiz hasta las hojas.

2. Mapa de tesoros

Un mapa de tesoros es un árbol con bifurcaciones que terminan en cofres. Cada bifurcación y cada cofre tiene un objeto, que puede ser chatarra o un tesoro.

```
data Dir = Izq | Der

data Objeto = Tesoro | Chatarra

data Mapa = Cofre Objeto
          | Bifurcacion Objeto Mapa Mapa
```

Definir las siguientes operaciones:

1. `hayTesoro :: Mapa -> Bool`
   Indica si hay un tesoro en alguna parte del mapa.
2. `hayTesoroEn :: [Dir] -> Mapa -> Bool`
   Indica si al final del camino hay un tesoro. Nota: el final del camino es la lista vacía de direcciones.
3. `caminoAlTesoro :: Mapa -> [Dir]`
   Indica el camino al tesoro. Precondición: hay un sólo tesoro en el mapa.
4. `caminoRamaMasLarga :: Mapa -> [Dir]`
   Indica el camino de la rama más larga.
5. `tesorosPerLevel :: Mapa -> [[Objecto]]`
   Devuelve los tesoros separados por nivel en el árbol.
6. `todosLosCaminos :: Mapa -> [[Dir]]`
   Devuelve todos lo caminos en el mapa.

Anexo con ejercicios adicionales

3. Expresiones aritméticas

Sea el tipo `Exp`, modelando expresiones aritméticas:

```
data Exp = Constante Int
         | ConsExpUnaria  OpUnaria  Exp
         | ConsExpBinaria OpBinaria Exp Exp

data OpUnaria = Neg

data OpBinaria = Suma | Resta | Mult | Div
```

Implementar las siguientes funciones:

1. `eval :: Exp -> Int`
   Dada una expresión evalúe esta expresión y retorne su valor. ¿Qué casos hacen que `eval` sea una función parcial?

2. `simplify :: Exp -> Exp`
   Dada una expresión la simplifica según los siguientes criterios:

   a) $0 + x = x + 0 = x$
   b) $x - 0 = x$
   c) $0 - x = -x$
   d) $x \times 1 = 1 \times x = x$
   e) $x \times 0 = 0 \times x = 0$
   f) $x \div 1 = x$
   g) $0 \div x = 0,\ x \neq 0$