Práctica de ejercicios #3 - Tipos recursivos

Estructuras de Datos, Universidad Nacional de Quilmes

12 de abril de 2023

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 videos publicados o clases presenciales****, 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.*

1. Tipos recursivos simples

1.1. Celdas con bolitas

Representaremos una celda con bolitas de colores rojas y azules, de la siguiente manera:

```
data Color = Azul | Rojo
data Celda = Bolita Color Celda | CeldaVacia
```

En dicha representación, la cantidad de apariciones de un determinado color denota la cantidad de bolitas de ese color en la celda. Por ejemplo, una celda con 2 bolitas azules y 2 rojas, podría ser la siguiente:

```
Bolita Rojo (Bolita Azul (Bolita Rojo (Bolita Azul CeldaVacia)))
```

Implementar las siguientes funciones sobre celdas:

- `nroBolitas :: Color -> Celda -> Int`
  Dados un color y una celda, indica la cantidad de bolitas de ese color. Nota: pensar si ya existe una operación sobre listas que ayude a resolver el problema.
- `poner :: Color -> Celda -> Celda`
  Dado un color y una celda, agrega una bolita de dicho color a la celda.
- `sacar :: Color -> Celda -> Celda`
  Dado un color y una celda, quita una bolita de dicho color de la celda. Nota: a diferencia de Gobstones, esta función es total.
- `ponerN :: Int -> Color -> Celda -> Celda`
  Dado un número n, un color c, y una celda, agrega n bolitas de color c a la celda.

1.2. Camino hacia el tesoro

Tenemos los siguientes tipos de datos

```
data Objeto = Cacharro | Tesoro
data Camino = Fin | Cofre [Objeto] Camino | Nada Camino
```

Definir las siguientes funciones:

- `hayTesoro :: Camino -> Bool`
  Indica si hay un cofre con un tesoro en el camino.
- `pasosHastaTesoro :: Camino -> Int`
  Indica la cantidad de pasos que hay que recorrer hasta llegar al primer cofre con un tesoro. Si un cofre con un tesoro está al principio del camino, la cantidad de pasos a recorrer es 0. Precondición: tiene que haber al menos un tesoro.
- `hayTesoroEn :: Int -> Camino -> Bool`
  Indica si hay un tesoro en una cierta cantidad exacta de pasos. Por ejemplo, si el número de pasos es 5, indica si hay un tesoro en 5 pasos.
- `alMenosNTesoros :: Int -> Camino -> Bool`
  Indica si hay al menos "n" tesoros en el camino.
- (desafío) `cantTesorosEntre :: Int -> Int -> Camino -> Int`
  Dado un rango de pasos, indica la cantidad de tesoros que hay en ese rango. Por ejemplo, si el rango es 3 y 5, indica la cantidad de tesoros que hay entre hacer 3 pasos y hacer 5. Están incluidos tanto 3 como 5 en el resultado.

2. Tipos arbóreos

2.1. Árboles binarios

Dada esta definición para árboles binarios

```
data Tree a = EmptyT | NodeT a (Tree a) (Tree a)
```

defina las siguientes funciones utilizando recursión estructural según corresponda:

1. `sumarT :: 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. `mapDobleT :: Tree Int -> Tree Int`
   Dado un árbol de enteros devuelve un árbol con el doble de cada número.
4. `perteneceT :: 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.
5. `aparicionesT :: 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.
6. `leaves :: Tree a -> [a]`
   Dado un árbol devuelve los elementos que se encuentran en sus hojas.

7. `heightT :: Tree a -> Int`
   Dado un árbol devuelve su altura.
   Nota: la altura de un árbol (height en inglés), también llamada profundidad, es la cantidad de niveles del árbol[1]*. La altura para EmptyT es 0, y para una hoja es 1.*

8. `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.

9. `toList :: 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.

10. `levelN :: Int -> Tree a -> [a]`
    Dados un número n y un árbol devuelve una lista con los nodos de nivel n. El nivel de un nodo es la distancia que hay de la raíz hasta él. La distancia de la raiz a sí misma es 0, y la distancia de la raiz a uno de sus hijos es 1.
    Nota: El primer nivel de un árbol (su raíz) es 0.

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

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

13. `todosLosCaminos :: Tree a -> [[a]]`
    Dado un árbol devuelve todos los caminos, es decir, los caminos desde la raíz hasta cualquiera de los nodos.

```
todosLosCaminos (NodeT 1 (NodeT 2 (NodeT 3 EmptyT EmptyT)
                                  EmptyT)
                         (NodeT 4 (NodeT 5 EmptyT EmptyT)
                                   EmptyT))
 = [ [1], [1,2], [1,2,3], [1,4], [1,4,5] ]
```

2.2. Expresiones Aritméticas

El tipo algebraico `ExpA` modela expresiones aritméticas de la siguiente manera:

```
data ExpA = Valor Int
          | Sum  ExpA ExpA
          | Prod ExpA ExpA
          | Neg  ExpA
```

Implementar las siguientes funciones utilizando el esquema de recursión estructural sobre `Exp`:

1. `eval :: ExpA -> -> Int`
   Dada una expresión aritmética devuelve el resultado evaluarla.

2. `simplificar :: ExpA -> ExpA`
   Dada una expresión aritmética, la simplifica según los siguientes criterios (descritos utilizando notación matemática convencional):

[1]También existen otras definiciones posibles. Por ejemplo, puede definirse como la distancia del camino desde la raíz a su hoja más lejana. Por distancia entendemos la cantidad de nodos que hay en dicho camino. En este caso las hojas tendrían altura 0, porque la distancia del camino a sí mismos lo es. Se suele utilizar más en árboles que no poseen un constructor vacío.

a) `0 + x = x + 0 = x`

b) `0 * x = x * 0 = 0`

c) `1 * x = x * 1 = x`

d) `- (- x) = x`