Programación Funcional

Ejercicios de Práctica Nro. 9

Inducción/Recursión III

Aclaraciones:

- Los ejercicios siguen un orden de complejidad creciente, y cada uno puede servir a los siguientes. No se recomienda saltear ejercicios sin consultar antes a un docente.
- Recordar que se pueden aprovechar en todo momento las funciones ya definidas, tanto las de esta misma práctica como las de prácticas anteriores.
- Probar todas las 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 evalúan principalmente este aspecto. Para utilizando formas alternativas al resolver los ejercicios consultar a los docentes.

Ejercicio 1) Dada la siguiente representación de expresiones aritméticas

```
data EA = Const Int | BOp BinOp EA EA
data BinOp = Sum | Mul
```

a. implementar las siguientes funciones por recursión estructural (o primitiva en caso de ser necesario):

   i. **`evalEA :: EA -> Int`**, que describe el número que resulta de evaluar la cuenta representada por la expresión aritmética dada.
   ii. **`ea2ExpA :: EA -> ExpA`**, que describe una expresión aritmética representada con el tipo **`ExpA`**, cuya estructura y significado son los mismos que la dada.
   iii. **`expA2ea :: ExpA -> EA`**, que describe una expresión aritmética representada con el tipo **`EA`**, cuya estructura y significado son los mismos que la dada.

b. demostrar la siguiente propiedad:

   i. **`ea2ExpA . expA2ea = id`**
   ii. **`expA2ea . ea2ExpA = id`**
   iii. **`evalExpA . ea2ExpA = evalEA`**
   iv. **`evalEA . expA2ea = evalExpA`**

Ejercicio 2) Dada la siguiente definición

```
data Arbol a b = Hoja b | Nodo a (Arbol a b) (Arbol a b)
```

que representa una estructura de árboles binarios;

a. implementar las siguientes funciones por recursión estructural (o primitiva en caso de ser necesario):

i. **`cantidadDeHojas :: Arbol a b -> Int`**, que describe la cantidad de hojas en el árbol dado.
ii. **`cantidadDeNodos :: Arbol a b -> Int`**, que describe la cantidad de nodos en el árbol dado.
iii. **`cantidadDeConstructores :: Arbol a b -> Int`**, que describe la cantidad de constructores en el árbol dado.
iv. **`ea2Arbol :: EA -> Arbol BinOp Int`**, que describe la representación como elemento del tipo **`Arbol BinOp Int`** de la expresión aritmética dada.

b. demostrar la siguiente propiedad:

para todo `t :: Arbol a b`

```
cantidadDeHojas t + cantidadDeNodos t
  = cantidadDeConstructores t
```

Ejercicio 3) Dada la siguiente definición

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

que expresa la estructura de árboles binarios;

a. implementar las siguientes funciones utilizando recursión estructural (o primitiva en caso de ser necesario):
   i. **`sumarT :: Tree Int -> Int`**, que describe el número resultante de sumar todos los números en el árbol dado.
   ii. **`sizeT :: Tree a -> Int`**, que describe la cantidad de elementos en el árbol dado.
   iii. **`anyT :: (a -> Bool) -> Tree a -> Bool`**, que indica si en el árbol dado hay al menos un elemento que cumple con el predicado dado.
   iv. **`countT :: (a -> Bool) -> Tree a -> Int`**, que describe la cantidad de elementos en el árbol dado que cumplen con el predicado dado.
   v. **`countLeaves :: Tree a -> Int`**, que describe la cantidad de hojas del árbol dado.
   vi. **`heightT :: Tree a -> Int`**, que describe la altura del árbol dado.
   vii. **`inOrder :: Tree a -> [a]`**, que describe la lista *in order* con los elementos del árbol dado.
   viii. **`listPerLevel :: Tree a -> [[a]]`**, que describe la lista donde cada elemento es una lista con los elementos de un nivel del árbol dado.
   ix. **`mirrorT :: Tree a -> Tree a`**, que describe un árbol con los mismos elemento que el árbol dado pero en orden inverso.
   x. **`levelN :: Int -> Tree a -> [a]`**, que describe la lista con los elementos del nivel dado en el árbol dado.
   xi. **`ramaMasLarga :: Tree a -> [a]`**, que describe la lista con los elementos de la rama más larga del árbol.

xii. `todosLosCaminos :: Tree a -> [[a]]`, que describe la lista con todos los caminos existentes en el árbol dado.

b. demostrar las siguientes propiedades:
   i. `heightT = length . ramaMasLarga`
   ii. `reverse . listInOrder = listInOrder . mirrorT`

Ejercicio 4) Dada la siguiente definición

```
data AppList a = Single a
               | Append (AppList a) (AppList a)
```

cuya intención es describir una representación no lineal de listas no vacías;

a. implementar las siguientes funciones por recursión estructural (o primitiva en caso de ser necesario):
   i. `lenAL :: AppList a -> Int`, que describe la cantidad de elementos de la lista.
   ii. `consAL :: a -> AppList a -> AppList a`, que describe la lista resultante de agregar el elemento dado al principio de la lista dada.
   iii. `headAL :: AppList a -> a`, que describe el primer elemento de la lista dada.
   iv. `tailAL :: AppList a -> AppList a`, que describe la lista resultante de quitar el primer elemento de la lista dada.
   v. `snocAL :: a -> AppList a -> AppList a`, que describe la lista resultante de agregar el elemento dado al final de la lista dada.
   vi. `lastAL :: AppList a -> a`, que describe el último elemento de la lista dada.
   vii. `initAL :: AppList a -> AppList a`, que describe la lista dada sin su último elemento.
   viii. `reverseAL :: AppList a -> AppList a`, que describe la lista dada con sus elementos en orden inverso.
   ix. `elemAL :: Eq a => a -> AppList a -> Bool`, que indica si el elemento dado se encuentra en la lista dada.
   x. `appendAL :: AppList a -> AppList a -> AppList a`, que describe el resultado de agregar los elementos de la primera lista adelante de los elementos de la segunda.
   NOTA: buscar la manera más eficiente de hacerlo.
   xi. `appListToList :: AppList a -> [a]`, que describe la representación lineal de la lista dada.

b. demostrar las siguientes propiedades:
   i. para todo `xs :: AppList a`. para todo `ys :: AppList a`.
   `lenAL (appendAL xs ys) = lenAL xs + lenAL ys`
   ii. `reverseAL . reverseAL = id`
   iii. `(reverseAL .) . flip consAL . reverseAL = snocAL`

Ejercicio 5) Dadas las siguientes definiciones

```
data QuadTree a = LeafQ a
                | NodeQ (QuadTree a) (QuadTree a)
                        (QuadTree a) (QuadTree a)
data Color = RGB Int Int Int
type Image = QuadTree Color
```

donde **`QuadTree`** representa a la estructura de los árboles cuaternarios, **`Color`** representa a los colores con precisión *TrueColor*, e **`Image`** representa imágenes cuadradas de tamaños arbitrarios;

a. implementar las siguientes funciones por recursión estructural (o primitiva en caso de ser necesario):
   i. **`heightQT :: QuadTree a -> Int`**, que describe la altura del árbol dado.
   ii. **`countLeavesQT :: QuadTree a -> Int`**, que describe la cantidad de hojas del árbol dado.
   iii. **`sizeQT :: QuadTree a -> Int`**, que describe la cantidad de constructores del árbol dado.
   iv. **`compress :: QuadTree a -> QuadTree a`**, que describe el árbol resultante de transformar en hoja todos aquellos nodos para los que se cumpla que todos los elementos de sus subárboles son iguales.
   v. **`uncompress :: QuadTree a -> QuadTree a`**, que describe el árbol resultante de transformar en nodo (manteniendo el dato de la hoja correspondiente) todas aquellas hojas que no se encuentren en el nivel de la altura del árbol.
   vi. **`render :: Image -> Int -> Image`**, que describe la imagen dada en el tamaño dado.
   Precondición: el tamaño dado es potencia de 4 y es mayor o igual a la altura del árbol dado elevado a la 4ta potencia.
   NOTA: Una imagen tiene tamaño **`t`** cuando todas las hojas se encuentran en el nivel $log_4(\mathbf{t})$.
   AYUDA: Esta operación es similar a **`uncompress`**, pero pudiendo variar la altura del árbol

b. demostrar las siguientes propiedades:
   i. **`countLeavesQT . uncompress = (4^) . heightQT`**
   ii. **`uncompress = subst render heightQT`**