

# Trabajo Práctico # 3

Estructuras de Datos, Universidad Nacional de Quilmes

9 de septiembre de 2014

## 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 comprensión 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. `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. `mapOpuestoT :: Tree Dir -> Tree Dir`  
Dado un árbol de direcciones  $t$  devuelve un árbol con la dirección opuesta para cada elemento de  $t$ .  
*Nota: Utilizar el tipo Dir ya definido.*
5. `mapLongitudT :: Tree String -> Tree Int`  
Dado un árbol de palabras devuelve un árbol con la longitud de cada palabra.
6. `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.
7. `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  $e$ .

8. `promedioEdadesT :: Tree Persona -> Int`  
Dado un árbol de personas devuelve el promedio entre las edades de todas las personas.  
Definir las subtarear que sean necesarias para resolver esta función.  
*Nota: Utilizar el tipo Persona ya definido.*
9. `engancharYSumaEnRaiz :: Tree Int -> Tree Int -> Tree Int`  
Dados dos árboles construye un árbol  $t$  en el que ambos árboles son hijos de  $t$ , y en la raíz de  $t$  se guarda la suma de todos los elementos de los hijos de  $t$ . ¿Se utiliza recursión para definir esta función?
10. `leaves :: Tree a -> Int`  
Dado un árbol devuelve su cantidad de hojas.  
*Nota: una hoja (leaf en inglés) es un nodo que no tiene hijos.*
11. `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.*
12. `nodes :: 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.
13. `espejoT :: Tree a -> Tree a`  
Dado un árbol devuelve el árbol resultante de intercambiar el hijo izquierdo con el derecho, en cada nodo del árbol.
14. `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 raíz y luego los elementos del hijo derecho.*
15. `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 raíz, luego los elementos del hijo izquierdo, a continuación los elementos del hijo derecho.*
16. `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 raíz.*
17. `concatenarListasT :: Tree [a] -> [a]`  
Dado un árbol de listas devuelve la concatenación de todas esas listas. El recorrido debe ser *in-order*.
18. `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.*
19. `listPerLevel :: Tree a -> [[a]]`  
Dado un árbol devuelve una lista de listas en la que cada elemento representa un nivel de dicho árbol.
20. `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.

## 2. Expresiones aritméticas

### Ejercicio 1

Sea el tipo `Exp`, modelando expresiones aritméticas:

BinOp por OpBinaria UnOp por OpUnaria  
 BinExp por ConsExpBinaria UnExp por ConsExpUnaria  
 Const por Constante

```
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. `simplificar :: 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)  $0 \div x = 0, x \neq 0$

### Ejercicio 2

Agregamos ahora dos nuevos operadores unarios a nuestro tipo de expresiones:

```
data OpUnaria = Neg | Inc | Dec
```

Modificar `eval` y `simplificar` incluyendo el manejo de estas nuevas operaciones unarias (*Inc* y *Dec* significan incrementar y decrementar respectivamente). Para `simplificar`, considerar estos criterios de simplificación adicionales:

1. Reemplazar  $x + 1 = 1 + x$  por `Inc x`
2. Reemplazar  $x - 1$  por `Dec x`
3. Reemplazar  $1 - x$  por `-(Dec x)`

## Anexo con ejercicios adicionales

Utilizando los tipos de datos ya definidos en clase, implementar las siguientes funciones:

1. `cantidadDePokemonesDe :: TipoDePokemon -> Tree Pokemon -> Int`  
Dados un tipo de pokemon y un árbol de pokemons devuelve la cantidad de pokemons de ese tipo en el árbol.
2. `entrenadoresExpertos :: Tree Entrenador -> [Entrenador]`  
Dado un árbol de entrenadores pokemon devuelve la lista con todos los entrenadores considerados expertos. Un entrenador es experto si posee al menos un pokemon de cada tipo posible.
3. `concatenarArboles :: [Tree a] -> [a]`  
Dada una listas de árboles devolver la lista con todos los elementos de todos los árboles. No importa el orden de los elementos en la lista resultante.