# Práctica 4

Programación Funcional, UNQ

Tipos inductivos y recursión

***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. Listas

Para las funciones del ejercicio 5 de la práctica 1 (funciones sobre listas), determine:

a) Cuáles definió por recursión estructural.

b) Cuáles definió por recursión a la cola.

c) Cuáles definió por recursión lineal.

d) Cuáles siempre terminan (justifique).

## 2. Árboles

Dada la siguiente definición para árboles (que sólo contiene datos en las hojas):

```
data TipTree a = Tip a | Join (TipTree a) (TipTree a)
```

De el tipo y defina las siguientes funciones:

a) `heightTip` que devuelve la longitud del camino más largo desde la raíz hasta una hoja.

b) `leaves` que calcula el número de hojas.

c) `nodes` que calcula en número de nodos que no son hojas.

d) `walkover` que devuelve la lista de las hojas de un árbol, leídas de izquierda a derecha.

e) `mirrorTip` que calcula la imagen especular del árbol, o sea, el árbol obtenido intercambiando los subárboles izquierdo y derecho de cada nodo.

f) `mapTip` que toma una función y un árbol, y devuelve el árbol que se obtiene de aplicar la función al valor almacenado en cada hoja.

# 3. Polinomios

Considere la siguiente representación de polinomios con coeficientes enteros:

```
data Poli = Cte Int | Var | Add Poli Poli | Mul Poli Poli
```

Con la que el polinomio $P(x) = x^2 + 3x + 5$ puede representarse de la siguiente manera:

```
Add (Mul Var Var) (Add (Mul (Cte 3) Var) Cte 5)
```

Escriba las siguientes funciones:

a) `eval :: Poli -> Int -> Int`
que retorna el resultado de evaluar un polinomio $P$ con un valor $x$ dado ($P(x)$).

b) `mEscalar :: Poli -> Int -> Poli`
que retorna el polinomio obtenido luego de multiplicar cada constante y variable por un valor entero.

c) `sOptimize :: Poli -> Poli`
que retorna un polinomio equivalente al tomado como parámetro pero donde las operaciones de suma y multiplicación entre constantes ya fueron resueltas, es decir, un polinomio en donde no existen constructores de la forma `Add (Cte _) (Cte _)` ni `Mul (Cte _) (Cte _)`

# 4. Fórmulas lógicas

Considere la siguiente representación de expresiones lógicas:

```
type Variable = Int  -- Identificadores enteros para variables

data Logical = Var Variable         | -- una variable con un id dado
               Not Logical          | -- la negacion de una expresion
               And Logical Logical  | -- la conjuncion de expresiones
               Or  Logical Logical    -- la disyuncion de expresiones
```

Por ejemplo la expresión $(x_1 \vee x_2) \wedge (\neg x_3 \vee x_4)$ se puede escribir como:
`And (Or (Var 1) (Var 2)) (Or (Not (Var 3)) (Var 4))`

Observe que estas expresiones no contienen constantes, para evaluarlas es necesario contar con una valuación:

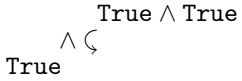
```
type Valuation = Variable -> Bool
```

Una valuación es una función de `Variable` en `Bool`. Es decir, una valuación asigna a cada variable el valor verdadero o falso.
Por ejemplo sea `v` una función tal que:

```
v 1 = True
v 2 = True
v x = False -- para todo x distinto de 1 y 2
```

Al evaluar la expresión anterior con esta valuación se obtiene verdadero pues,

$$(x_1 \vee x_2) \wedge (\neg x_3 \vee x_4)$$
$$\overset{v \curvearrowright}{=} (\texttt{True} \vee \texttt{True}) \wedge (\neg \texttt{False} \vee \texttt{False})$$
$$\overset{\neg \curvearrowright}{=} (\texttt{True} \vee \texttt{True}) \wedge (\texttt{True} \vee \texttt{False})$$
$$\overset{\vee \curvearrowright}{=}$$




Defina las siguientes funciones:

a) `eval :: Logical -> Valuation -> Bool`
Que retorna el resultado de resolver la expresión lógica con la valuación dada.

b) `vars :: Logical -> [Int]`
Que retorna la lista de todas las variables con ocurrencias en una expresión dada.

c) `simp :: Logical -> Logical`
Que simplifica las expresiones eliminando operaciones triviales, más específicamente la doble negación y la conjunción o disyunción de variables iguales.

# 5. Secuencias

Considere la siguiente representación de secuencias:

```
data Seq a = Nil | Unit a | Cat (Seq a) (Seq a)
```

El constructor `Nil` representa una secuencia vacía. `Unit x` representa una secuencia unitaria, cuyo único elemento es `x`. Finalmente, `Cat x y` representa una secuencia cuyos elementos son todos los de la secuencia `x`, seguidos por todos los de la secuencia `y`.

Definir las siguientes operaciones:

a) `appSeq :: Seq a -> Seq a -> Seq a`
que toma dos secuencias y devuelve su concatenación.

b) `conSeq :: a -> Seq a -> Seq a`
que toma un elemento y una secuencia y devuelve la secuencia que tiene al elemento dado como cabeza y a la secuencia dada como cola.

c) `lenSeq :: Seq a -> Int`
que calcula la cantidad de elementos de una secuencia.

d) `revSeq :: Seq a -> Seq a`
que toma una secuencia e invierte sus elementos.

e) `headSeq :: Seq a -> a`
que toma una secuencia y devuelve su primer elemento (es decir el de más a la izquierda).

f) `tailSeq :: Seq a -> Seq a`
que remueve la cabeza de una secuencia.

g) `normSeq :: Seq a -> Seq a`
que elimina todos los `Nil`s innecesarios de una secuencia.
Por ejemplo, `normSeq (Cat (Cat Nil (Unit 1)) Nil) = Unit 1`

h) `eqSeq :: Seq a -> Seq a -> Bool`
que toma dos secuencias y devuelve `True` si ambas contienen los mismos valores, en el mismo orden y en la misma cantidad.

i) `seq2List :: Seq a -> [a]`
que toma una secuencia y devuelve una lista con los mismos elementos, en el mismo orden.

j) ¿Qué ventajas y desventajas encuentra sobre `(Seq a)` respecto a las listas de Haskell (`[a]`)?

# 6. Árboles Generales

Dado el siguiente tipo para árboles generales (árboles con una cantidad arbitraria de hijos en cada nodo):

```
data GenTree a = GNode a [GenTree a]
```

Definir las siguientes funciones:

a) `sizeGT :: GenTree a -> Int`
que retorna la cantidad de elementos en el árbol.

b) `heightGT :: GenTree a -> Int`
que retorna la altura del árbol.

c) `mirrorGT :: GenTree a -> GenTree a`
que calcula la imagen especular del árbol.

d) `toListGt :: GenTree a -> [a]`
que retorna una lista con los elementos en el árbol.

e) `mapGT :: (a -> b) -> GenTree a -> GenTree b`
que aplica una función dada a cada elemento en el árbol retornando uno estructuralmente equivalente.

f) `levelNGT :: GenTree a -> Int -> [a]`
que retorna todos los elementos en un nivel dado del árbol.

# 7. Árboles de Bits

Llamaremos árbol de bits (`BitTree`) a una estructura de datos utilizada para representar conjuntos de elementos que pueden ser representados como secuencias de bits (e.g. `Int`). Cada subárbol izquierdo contiene los elementos cuya representación en secuencia de bits comienza con un cero (`Z`), mientras que un subárbol derecho contiene los elementos que comienzan con un uno (`O`). Para verificar si un elemento está contenido en la estructura es necesario recorrer una rama del árbol hasta agotar su representación como secuencia de bits. Si el nodo alcanzado contiene un `True`, entonces el elemento ésta, en caso contrario no está.

```
data Bit = Z | O

data BitTree = Nil | Bin Bool BitTree BitTree
```

Tenga en cuenta que la estructura debe respetar el invariante de representación que toda rama debe terminar en un nodo con `True` (ya que en caso contrario podría reemplazarse con `Nil`). Y defina las siguientes funciones:

a) `contains :: [Bit] -> BitTree -> Bool`, que indica si un elemento (representado como una secuencia de bits) está contenido en la estructura.

b) `insert :: [Bit] -> BitTree -> BitTree`, que inserta un elemento (representado como una secuencia de bits) en un árbol de bits (i.e., `contains bs (insert bs Nil) == True`).

c) `remove :: [Bit] -> BitTree -> BitTree`, que elimina un elemento de un árbol de bits (tenga en cuenta que en este caso puede ser necesario restablecer el invariante de representación).

d) `union :: BitTree -> BitTree -> BitTree`, que retorna uno árbol de bits que contiene la unión de los elementos en dos árboles dados.

e) `intersect :: BitTree -> BitTree -> BitTree`, que retorna un árbol de bits que contiene la intersección de los elementos en dos árboles dados.