

# PROGRAMACIÓN FUNCIONAL

## Trabajo Práctico Nro. 6

**Temas:** Sinónimos de tipos. Tipos algebraicos recursivos. Árboles.

### Bibliografía relacionada:

- ▮ Simon Thompson. The craft of Functional Programming. Addison Wesley, 1996. Cap. 4.
- L.C. Paulson. ML for the working programmer. Cambridge University Press, 1996. Cap. 4 y 7.
- Bird, Richard. Introduction to functional programming using Haskell. Prentice Hall, 1998 (Second Edition). Cap. 3 y 6.

1. Definir las operaciones de unión e intersección de dos conjuntos y el predicado de pertenencia para conjuntos de elementos de tipo `a`, los cuales se representan:

- a)* por extensión (como lista de elementos de `a`).
- b)* por comprensión (como predicado `a -> Bool`);

¿Qué ventajas encuentra a cada una de las representaciones?

2. Dado el tipo `data TipTree a = Tip a | Join (TipTree a) (TipTree a)`

Definir y dar el tipo de las siguientes funciones:

- `heightTip`, que devuelve la longitud del camino más largo desde la raíz hasta una hoja.
- `leaves`, que calcula el número de hojas.
- `nodes`, que calcula el número de nodos que no son hojas.
- `walkover`, que devuelve la lista de las hojas de un árbol, leídas de izquierda a derecha.
- `mirrorTip`, que calcula la imagen especular del árbol, o sea, el árbol obtenido intercambiando los subárboles izquierdo y derecho de cada nodo.
- `mapTip`, que toma una función y un árbol, y devuelve el árbol que se obtiene del dado al aplicar la función a cada nodo.

3. ¿Se pueden representar listas ordenadas mediante tipos algebraicos? Proponga una definición o justifique.

4.  $\odot$  Considere el tipo 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`.

a) Definir las siguientes operaciones:

<code>appSeq,</code>	que toma dos secuencias y devuelve su concatenación.
<code>conSeq,</code>	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.
<code>lenSeq,</code>	que calcula la cantidad de elementos de una secuencia.
<code>revSeq,</code>	que toma una secuencia e invierte sus elementos.
<code>headSeq,</code>	que toma una secuencia y devuelve su primer elemento (es decir el de más a la izquierda).
<code>tailSeq,</code>	que remueve la cabeza de una secuencia.
<code>normSeq,</code>	que elimina todos los <code>Nils</code> innecesarios de una secuencia. Por ejemplo, <code>normSeq (Cat (Cat Nil (Unit 1)) Nil) = Unit 1</code>
<code>eqSeq,</code>	que toma dos secuencias y devuelve <code>True</code> si ambas contienen los mismos valores, en el mismo orden y en la misma cantidad.
<code>seq2List,</code>	que toma una secuencia y devuelve una lista con los mismos elementos, en el mismo orden.

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

### Ejercicios complementarios

5. Son conocidas en Lógica las siguientes identidades:

$$\begin{aligned}
 p \wedge q &\equiv \neg(\neg p \vee \neg q) & p \rightarrow q &\equiv \neg p \vee q & p \leftrightarrow q &\equiv \neg(\neg(\neg p \vee q) \vee \neg(\neg q \vee p)) \\
 (\forall x)p &\equiv \neg(\exists x)(\neg p)
 \end{aligned}$$

Dado el siguiente tipo, que representa de manera simplificada las fórmulas de la lógica:

```

data Form = Atom                | Not Form
           | Or Form Form        | And Form Form
           | Implies Form Form   | Iff Form Form
           | Forall Var Form     | Exists Var Form

```

- a) Definir una función `normalize :: Form -> Form` que dada una expresión del tipo `Form` retorne otra del mismo tipo que utilice solamente los conectivos  $\neg$  y  $\vee$  y el cuantificador existencial, y sea equivalente a la dada. Ejemplo:
- ```
normalize ((Implies (Not Atom) (And (Forall "x" Atom) Atom)))
```
- da como resultado

```
Or (Not (Not Atom))
   (Not (Or (Not (Not (Exists "x" (Not Atom)))) (Not Atom)))
```

- b) Definir un tipo algebraico `FN` que sirva para representar expresiones lógicas en forma normal, y las funciones:

`fn2FN`, que dado un objeto de tipo `Form` en forma normal, lo transforma en uno de tipo `FN` equivalente.

`form2FN`, que dado un objeto cualquiera de tipo `Form`, lo transforma en uno de tipo `FN` equivalente.