Fundamentos de la Computación
Parcial

A no ser que la letra indique lo contrario, toda función que se use o pida debe ser definida (lo cual incluye declarar su tipo), y todo lema utilizado debe ser demostrado. En todos los ejercicios podrá hacer uso de los siguientes conectivos booleanos:

```
(&&) :: Bool -> Bool -> Bool
(&&) = \x y -> case x of {True -> y ; False -> False}
(||) :: Bool -> Bool -> Bool
(||) = \x y -> case x of {True -> True ; False -> y}
not :: Bool -> Bool
not = \x -> case x of {True -> False ; False -> True}
```

Problema 0. `[Defensa]` Considere:

```
swap :: Prog
swap =  Asig "aux" (V"x")
       :> Asig "x" (V"y")
       :> Asig "y" (V"aux")

m :: Mem
m = [("x", 14), ("y", 7)]
```

¿Qué resultado se obtiene al ejecutar `run swap m`?

Problema 1. `[20p]`

(a) Una lista `l1` es *prefijo* de otra lista `l2`, si existe una lista `l3` tal que `l1 ++ l3 = l2`.
Por ejemplo: `[1,2,3]` es prefijo de `[1,2,3,4,5]` y de `[1,2,3]`, pero no es prefijo de `[0,1,2,3,4]` ni de `[3,1,2]`.
Defina la función `prefijo::Eq a => [a] -> [a] -> Bool`, que determina si la primera lista es un prefijo de la segunda.

(b) Una lista `l1` es *sublista* de otra lista `l2`, si todos los elementos de `l1` aparecen en `l2`, en el mismo orden (`l2` puede tener más elementos eventualmente).
Por ejemplo: `[1,3,3]` es sublista de `[0,2,4,1,3,5,3,7]` y de `[1,1,2,2,3,3]`, pero no es sublista de `[3,1,1,3]` ni de `[1,1,3]`.
Defina la función `sublista::Eq a => [a] -> [a] -> Bool`, que determina si la primera lista es una sublista de la segunda.

Problema 2. `[20p]`

Considere la siguiente función `g :: X -> N -> N`:

```
g = \t n -> case t of { A -> case n of { 0 -> 0 ; S k -> S(g t k) } ;
                        B x y z -> case x of { True -> g y n ; False -> g z n } }
```

(a) Defina el tipo `X` para que la función `g` compile: `data X where {A ::...  ; B ::...  }`.

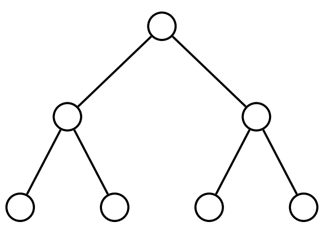
(b) Demuestre que `(∀t::X)(∀n::N) g t n = n`.

Problema 3. `[20p]` Considere el siguiente tipo `T` definido como:

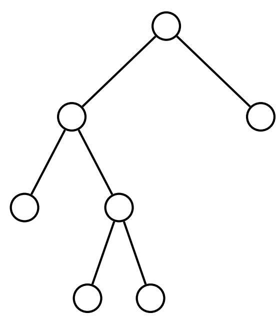
```
data T where { H ::  T ;
               I ::  T -> T -> T }
```

Un árbol T se llama *perfecto* si y sólo si todos sus niveles están completos.
Ejemplos:

Árbol perfecto de 3 niveles:



Árbol no perfecto de 4 niveles:



(a) Defina la función `niveles :: T -> N` que computa la cantidad de niveles de un árbol de tipo `T` (no necesariamente perfecto). Puede hacer uso de la función `(<=) :: N -> N -> N`.

(b) Defina la función `perfecto :: T -> Bool` que verifique si un árbol es perfecto. Puede hacer uso la función `(==) :: N -> N -> N`.

(c) Defina la función `perfGen :: N -> T`, que recibe un natural `n` y genera un árbol perfecto con `n + 1` niveles.

(d) Demuestre que que `perfGen` es correcta en relación a la definición de `perfecto`, es decir:
`(∀ n::N) perfecto (perfGen n) = True`.
Puede utilizar la reflexividad de `(==)` en `N` sin necesidad de demostrarla, o sea:
`(∀n::N) n == n = True`.