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]` Completar:

```
run ::  Prog -> Mem -> Mem
run = \p-> \m-> case p of{
                       Asig x e -> upd (x, eval e m) m
                     ; p1 :> p2 -> run p2 (run p1 m)
                     ; IF b p1 p2 -> case beval b m of {
                                         False -> run p2 m
                                       ; True -> run p1 m }
                     ; While b p1 -> ...  }
```

Problema 1. `[20p]` Una lista `l` es una *alternancia* de dos valores `x` e `y`, si y sólo si:

- Los elementos de `l` son exclusivamente `x` o `y` .
- En caso de que `l` no sea vacía, `x` e `y` aparecen en `l` en posiciones alternas, con `x` en la posición inicial.

Ejemplos: `[True, False, True, False, True]` es una alternancia de `True` y `False`.
`[False, True, False, True]` es una alternancia de `False` y `True`.
`[0,1,1,0,1]` no es una alternancia.
`[0]` es una alternancia de `0` y `1`.

(a) Defina utilizando recursión primitiva un predicado `alt` que reciba valores `x` e `y` y una lista `l`, y verifique si `l` es una *alternancia* de `x` e `y`.

(b) Defina utilizando recursión primitiva la función `altGen` que reciba un natural `n` y dos elementos `x` e `y` de cierto tipo `a`, y genere una alternancia de `x` e `y` de largo `n`.

Problema 2. [20p]

Considere la siguiente función:

```
f = \x y z -> case x of { [] -> y && z;
                          w:ws -> case w of { 0 -> case z of { True -> y ;
                                                               False -> False } ;
                                              S k -> f ws y z }}
```

donde `(&&)` es la conjunción booleana definida arriba.

(a) Dé el tipo de `f`.

(b) Demuestre que `(∀x::..)(∀y::..)(∀z::..)  f x y z = y && z`.
Puede asumir la conmutatividad y asociatividad de `(&&)` sin necesidad de demostrarlas.

Problema 3. [20p]

Considere el tipo `T a b`:

```
data T a b where { J :: a -> T a b ;
                   K :: b -> T a b ;
                   N :: T a b -> T a b -> T a b ;
                   P :: T a b -> T a b }
```

(a) Dibuje el árbol correspondiente a la siguiente expresión Haskell, y dé su tipo:
`P (N (J True) (N (P(K 2))(J False)))`

(b) Defina la función `numH :: T a b -> N` que calcule la cantidad de hojas de un árbol dado, puede utilizar la suma de naturales `(+)::N -> N -> N` definida en clase.

(c) Defina la función `numN :: T a b -> N` que calcule la cantidad de nodos `N` que tiene un árbol.

(d) Demuestre que `(∀t::T a b) numH t = S(numN t)`. Puede utilizar las propiedades de la suma de Naturales que considere necesarias, las cuales debe enunciar como lemas (sin necesidad de demostrarlos).