FUNDAMENTOS DE COMPUTACIÓN
PRÁCTICO 2
TIPOS FINITOS

1. Booleanos. En el Laboratorio 1 hay varios ejemplos de funciones Booleanas.
En este práctico se va a hacer énfasis en las demostraciones de propiedades de las mismas.

Ej.1. Considere los conectivos Booleanos definidos en clase:

```
not :: Bool -> Bool
not = \x -> case x of {False -> True ; True -> False}

(||) :: Bool -> Bool -> Bool
(||) = \x \y -> case x of {False -> y ; True -> True}

(&&) :: Bool -> Bool -> Bool
(&&) = \x y ->case x of {False -> False ; True -> y}
```

Demuestre las siguientes propiedades:

1) `(∀b::Bool) b && b = b`.
2) `(∀b::Bool) b || b = b`.
3) `(∀b::Bool) b && not b = False`
4) `(∀b::Bool) b || not b = True`
5) Conmutatividad de `(&&)`: `(∀b1::Bool,∀b2::Bool) b1 && b2 = b2 && b1`
6) Conmutatividad de `(||)`: `(∀b1::Bool,∀b2::Bool) b1 || b2 = b2 || b1`
7) Asociatividad de `(&&)`: `(∀b1,b2,b3::Bool) (b1 && b2) && b3 = b1 && (b2 && b3)`
8) Asociatividad de `(||)`: `(∀b1,b2,b3::Bool) (b1 || b2) || b3 = b1 || (b2 || b3)`
9) Distributividad de `(&&)` y `(||)`:
   `(∀b1,b2,b3::Bool) b1 && (b2 || b3) = (b1 && b2) || (b1 && b3)`
   `(∀b1,b2,b3::Bool) b1 || (b2 && b3) = (b1 || b2) && (b1 || b3)`
10) Leyes de De Morgan:
   `(∀b1,b2::Bool) not (b1 || b2) = not b1 && not b2`
   `(∀b1,b2::Bool) not (b1 && b2) = not b1 || not b2`

Ej.2. La equivalencia lógica (⇔) es la igualdad Booleana, y se define como el operador `==` en Haskell.

1) Defina `(==) :: Bool -> Bool -> Bool` usando case.
2) De otra definición de esta función usando las funciones del ejercicio anterior.

3) Demuestre que las dos definiciones anteriores son equivalentes, o sea:
`(∀b1,b2::Bool) b1 == b2 = b1 === b2`
4) Demuestre que `(==)` es conmutativa, o sea: `(∀b1,b2::Bool) b1 == b2 = b2 == b1`
5) Demuestre que `(==)` es asociativa, o sea:
`(∀b1,b2,b3::Bool) b1 == (b2 == b3) = (b1 == b2) == b3`

Ej.3.

1) Defina la función `(#) :: Bool -> Bool -> Bool`, que es `True` cuando exactamente uno de sus argumentos es `False`. Puede utilizar la función `not`.
2) Demuestre que `(∀b1,b2::Bool) not(b1 # b2) = (not b1) # b2`.
Puede utilizar sin demostrar la siguiente propiedad de `not`:
L$^{\texttt{not}}$: `(∀b::Bool) not (not b) = b`.
3) Demuestre que `(#)` es conmutativo.
4) Se cumple que `(#)` es asociativo?
    - En caso de responder afirmativamente, enuncie el lema correspondiente y demuéstrelo.
    - En caso de responder negativamente, alcanza con mostrar un contraejemplo (o sea, un caso en el que la propiedad no se cumple).

Ej.4.

1) Defina la función `(@) :: Bool -> Bool -> Bool`, que es `True` cuando por lo menos uno de sus argumentos es `False`. Puede utilizar la función `not`.
2) Demuestre que `(∀b1,b2::Bool) b1 @ b2 = not (b1 && b2)`.
3) ¿Se cumple que `(@)` es asociativo?
    - En caso de responder afirmativamente, enuncie el lema correspondiente y demuéstrelo.
    - En caso de responder negativamente, alcanza con mostrar un contraejemplo (o sea, valos específicos para los cuales la propiedad no se cumple).

Ej.5.

1) Defina la función `pares :: Bool -> Bool -> Bool -> Bool`, la cual recibe tres booleanos y devuelve `True` cuando una cantidad par de ellos es `True`.
2) Demuestre que `(∀b1,b2::Bool) par b1 b2 b1 = not b2`.

Ej.6.

1) Defina la función `mayoria :: Bool -> Bool -> Bool -> Bool`, la cual recibe tres booleanos y devuelve el resultado de la mayoría de ellos.
2) Demuestre que `(∀b1,b2::Bool) mayoria b1 b2 b1 = b1`.

2. Otros Tipos Finitos (Enumerados).

Ej.7.

1) Defina en Haskell el tipo `Part` para representar las partículas subatómicas Electrón, Neutrón y Protón.
2) Defina la función `inverso :: Part -> Part`, que retorne el opuesto de una partícula, considerando su carga. Recordar que el Electrón tiene carga negativa, el Neutrón no tiene carga y el Protón tiene carga positiva.
3) Defina la función `mayor :: Part -> Part -> Part`, que recibe dos partículas y devuelve aquella con mayor carga, consderando el siguiente orden: Electrón $<$ Neutrón $<$ Protón.

Ej.8.

1) Defina en Haskell el tipo `PC` para representar los puntos cardinales (Notre. Sur, Este y Oeste).
2) Defina la función `opuesto :: PC -> PC`, que retorne el opuesto de un punto cardinal.
3) Defina la función `siguiente :: PC -> PC`, que retorne el siguiente de un punto cardinal, en el sentido horario.
4) Demuestre que `(∀p::PC) opuesto (opuesto p) = p`.
5) Demuestre que `(∀p::PC) siguiente (siguiente p) = opuesto p`.

Ej.9. Se quiere jugar al juego de Piedra, Papel y Tijera. Éste es un juego de manos en el cual existen tres elementos: la *piedra* que vence a la tijera rompiéndola, la *tijera* que vencen al papel cortándolo, y el *papel* que vence a la piedra envolviéndola.
Se pide:

1) Defina en Haskell un tipo enumerado `PPT` que represente los tres elementos del juego.
2) Defina la función `gana::PPT-> PPT-> PPT` que recibe dos elementos y retorna el que gana. En caso de empate, deberá devolver el elemento recibido.
3) Demuestre que `(∀x::PPT) gana x x = x`.
4) Demuestre que `(∀x,y::PPT) gana x y = gana y x`.