FUNDAMENTOS DE LA COMPUTACIÓN
LABORATORIO 1 - BOOL

1. Para empezar

Vamos a trabajar en un editor de texto (cualquiera sirve), y ahí vamos a escribir nuestros progtramas Haskell, que se llaman *scripts* (guiones).
Para comenzar, escribir las siguientes líneas como primeras del programa:

```
{-#LANGUAGE GADTs, EmptyDataDecls, EmptyCase #-}
{-# OPTIONS_GHC -fno-warn-tabs #-}
```

La primera línea permite realizar definiciones de tipos con el formato visto en calse. La segunda línea es para poder usar caracteres de tabulación "tab" en el programa sin que el compilador genere *warnings* (mensajes de alerta).

2. Módulos

Lo que uno escribe en Haskell es en general un **módulo.** La siguiente línea será la que le da nombre al módulo. El nombre del módulo debe comenzar con mayúscula:

```
module Lab1 where
```

Finalmente, la siguiente línea permitirá importar del Preludio (la biblioteca estándar de Haskell) sólo la definición de la clase `Show`, que contiene funciones que permiten mostrar los resultados de nuestras expresiones en pantalla:

```
import Prelude (Show)
```

Las restantes definiciones del Preludio **no se importarán** para permitirnos efectuar definiciones que no se contradigan con las definidas en el Preludio de Haskell.

3. Funciones

Dado que para programar es preferible usar caracteres ASCII, Haskell no hace uso del carácter λ; en su lugar usa la barra inversa \.

4. Los Booleanos

' Para definir en Haskell el tipo de los Booleanos se utilizará la siguiente notación:

```
data Bool where {False::Bool ; True::Bool}
  deriving Show
```

La segunda línea (`deriving Show`) sirve sólo para que Haskell pueda mostrar los valores de `Bool` por pantalla.
Es importante ponerla con espacios adelante, ya que es la forma de indicarle al compilador que la misma forma parte de la definición del tipo. También podría ponerse en la misma línea de la definición:

```
data Bool where {False::Bool ; True::Bool}  deriving Show
```

Ejercicios

Ej. 1 Programar la negación booleana `not` y ponerla prueba con todos los casos posibles. ¿Cuántos son?

Ej. 2 Programar en Haskell los siguientes conectivos Booleanos y verificarlos en cada caso con todos los valores posibles (son 4 combinaciones en cada caso):

(a) Conjunción (y):
```
(&&) ::  Bool -> Bool -> Bool
(&&) = ....
```

(b) Disyunción (o inclusivo):
```
(||) ::  Bool -> Bool -> Bool
(||) = ....
```

(c) O exclusivo (sólo un argumento es `True`):
```
xor ::  Bool -> Bool -> Bool
xor = ...
```

(d) Ni (da `True` cuando ambos argumentos son `False`):
```
ni ::  Bool -> Bool -> Bool
ni = ...
```

Ej. 3 La equivalencia lógica ($\Leftrightarrow$) es la igualdad Booleana, y se define como el operador `==` en Haskell.

(a) Definir `(==) :: Bool -> Bool -> Bool` usando `case`.

(b) Dar otra definición de esta función usando otras funciones ya definidas y sin usar `case` (va a ser necesario darle otro nombre a la nueva función, por ejmplo `===`).

(c) Definir la desigualdad booleana `(/=) :: Bool -> Bool -> Bool`.
Compararla con el conectivo `xor`, el o exclusivo. ¿Qué conclusión se puede sacar?.

(d) Defina el orden entre booleanos como una función `(≤) :: Bool -> Bool -> Bool`, de modo tal que `False` sea menor que `True`.

Ej. 4 Programar las siguientes funciones booleanas. En todos los casos dar dos definiciones: una usando `case` y otra usando los conectivos definidos en los ejercicios anteriores (será necesario ponerle nombres diferentes a las funciones cuando se definan por segunda vez):

(a) `unanimidad :: Bool -> Bool -> Bool -> Bool`, la cual recibe tres booleanos y devuelve verdadero cuando los tres booleanos recibidos sean iguales.

(b) `mayoria :: Bool -> Bool -> Bool -> Bool`, la cual recibe tres booleanos y devuelve el resultado de la mayoría de ellos.

(c) `impar :: Bool -> Bool -> Bool -> Bool` tal que `impar b1 b2 b3` es `True` cuando una cantidad impar de sus argumentos es `True`.

Ej. 5 Redefinir los conectivos `(&&)`, `xor` y `ni` sin usar case, y utilizando sólo `not` y `(||)`.
Será necesario ponerle un nombre nuevo a las funciones para que no se choquen con los anteriores, por ejemplo `(&&&)`, `xxor` y `nii`.

Ej. 6 (a) Defina la función `(@@) :: Bool -> Bool -> Bool`, que es `True` cuando por lo menos uno de sus argumentos es `False`.

(b) Defina la función `(#) :: Bool -> Bool -> Bool`, que es `True` cuando exactamente uno de sus argumentos es `False`.

(c) Defina la función `tri :: Bool -> Bool -> Bool -> Bool`, tal que `tri b1 b2 b3` devuelve `True` si hay más argumentos `False` que `True`.