FUNDAMENTOS DE LA COMPUTACIÓN
LABORATORIO 1
3 DE ABRIL DE 2014

0. Preámbulo Lo que uno escribe en Haskell es en general un módulo. Por lo tanto, la primera línea será la que le da nombre al módulo. Este nombre debe ser el mismo que el del archivo que se está creando, comenzando con mayúscula:

```
module Lab1 where
```

La siguiente línea permitirá efectuar definiciones que no se contradigan con otras primitivas de Haskell:

```
import Prelude (Show)
```

1. Funciones
Dado que para programar es preferente usar caracteres ASCII, Haskell no hace uso del carcter λ, en su lugar usa la barra inversa `\`. Al mismo tiempo, se hace uso de `->` en vez de $\rightarrow$. Por último, se hace uso de `->` en lugar de `"."` para separar las variables abstraidas del cuerpo de una función.

Así, por ejemplo, podemos definir la función identidad en Haskell como:

```
id :: a -> a
id = \x -> x
```

?1. Poner a prueba la función `id` con distintas constantes numéricas

?2. Programar las siguientes funciones vistas en clase:

```
kid :: b -> a -> a
kid = ....

k :: a -> b -> a
k = ...

(.) :: (b ->c) -> (a->b) -> a -> c
(.) = ...

commute :: (a -> b -> c) -> (b -> a -> c)
commute = ...
```

?3. Definir el tipo de las siguientes funciones:

```
f1 :: ....
f1 = \f-> \x-> f x

f2 :: ....
f2 = \f-> \x-> f (f x)
```

```
f3 :: ....
f3 = \x-> \y -> \z -> z (y x) (y x)

f4 :: ....
f4 = \x-> \y -> \z -> z (y x x) x

f5 :: ....
f5 = \x-> \y -> \z -> z y y
```

?4. Encontrar expresiones que tengan los siguientes tipos:

```
h1 :: a -> (a -> b) -> b
h1 = ....

h2 :: (a -> b) -> (b -> c) -> a -> c
h2 = ....

h3 :: (a -> a -> b) -> a -> b
h3 = ....


h4 :: ((c -> b) -> b -> c -> a) -> c -> (c -> b) -> a
h4 = ....

h5 :: (a -> b -> a) -> b -> (b -> a) -> a
h5 = ....
```

2. Los Booleanos Comenzamos introduciendo el tipo de los booleanos en Haskell como sigue:

```
data Bool = False | True
  deriving (Show)
```

La segunda línea sirve sólo para que Haskell pueda mostrar esos valores por pantalla.

De inmediato programamos la negación:

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

?5. Poner a prueba la función `not` en todos los casos posibles

Más adelante nos hará falta también la disyunción lógica:

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

Los paréntesis alrededor del símbolo || son necesarios para usarlo en forma *infija*.
?6. Verificar la tabla de verdad de este conectivo

?7. Programar el resto de los conectivos Booleanos:

```
(&&) :: Bool -> Bool -> Bool
(&&) = ....

(>>) :: Bool -> Bool -> Bool
(>>) = ...

ni :: Bool -> Bool -> Bool
ni = ...
```

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

(1) Definir `(==) :: Bool -> Bool -> Bool` sin usar funciones auxiliares
(2) Dar otra definición de esta función sin usar `case` (va a ser necesario darle otro nombre a la nueva función)
(3) Definir la desigualdad booleana `(/=) :: Bool -> Bool -> Bool`. Compararla con el conectivo `xor`, la disyunción excluyente.