FUNDAMENTOS DE LA COMPUTACIÓN
LABORATORIO 2 - NÚMEROS NATURALES

Recuerden agregar las siguientes dos líneas al principio del programa:

```
{-#LANGUAGE GADTs #-}
{-# OPTIONS_GHC -fno-warn-tabs #-}
{-# OPTIONS_GHC -fno-warn-missing-methods #-}
```

La primera permite definir los tipos introducidos en el curso, la segunda evita que el compilador de errores por el uso de tabulaciones (tabs), y la tercera permite instanciar parcialmente algunas clases. Luego se debe comenzar el programa Haskell con la declaración del nombre del módulo, que debe ser el mismo que el del archivo (Naturales.hs).

```
module Naturales where

data N where { O :: N ; S :: N -> N } deriving Show
```

Nótese que por defecto estamos importando el Preludio de Haskell, que incluye los booleanos, todas sus funciones y otras definiciones que nos serán útiles más adelante.
Eso se realiza de forma automatica, a no ser que explícitamente especifiquemos qué funciones importar (como es el caso del uso que hicimos hasta ahora de: import Prelude (Show)).
Conviene definir algunos naturales y funciones para poder probar:

```
uno :: N
uno = S O

dos :: N
dos = S uno

tres :: N
tres = S dos

cuatro :: N
cuatro = S tres

cinco :: N
cinco = S cuatro

predecesor :: N -> N
predecesor = \n -> case n of {O -> O; S x -> x}
```

Clases de Haskell

Para poder utilizar los mismos nombres de funciones y operadores en distintos tipos, Haskell proporciona un modo estructurado de controlar esta sobrecarga (conocida como polimorfismo ad hoc).
Claramente, el comportamiento de las funciones y operadores sobrecargados suele ser distinto para cada tipo de dato. Un ejemplo de esto es la función `(==)` que debería tener tipo `t -> t -> Bool`, para cualquier tipo `t` donde se quiera definir una relación de igualdad.
Las clases Haskell resuelven este problemas del siguiente modo: Una clase (`class`) especifica un conjunto de funciones y operaciones con sus respectivos tipos.
Es posible luego declarar qué tipos son instancias de qué clases, y dar definiciones para las operaciones (sobrecargadas) asociadas con cada clase.

Por ejemplo, se podría definir la clase de tipos que tienen el operador de igualdad como sigue:

```
class Eq a where
  (==) :: a -> a -> Bool
```

Para definir que cierto tipo es instancia de una clase, se debe definir las funciones y operadores de la clase.

```
instance Eq N where
  (==) = ...
```

En realidad, la calse `Eq` también viene equipada con la desigualdad (`(/=)`), pero como ésta es la negación de la igualdad, se predefine en la misma clase.

Así, la definición de la clase `Eq` es la siguiente:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  (/=) = \ x y -> not (x == y)
```

Ej.1.

1) Defina la instancia de `Eq` para `N`:

```
instance Eq N where
  (==) = ...
```

2) Pruebe la función definida con algunos ejemplos. ¿Qué resultados dan las siguientes expresiones?: `predecesor tres == dos`, `triple dos == cuatro`, `cuadruple dos = doble cuatro`. También puede probar desigualdades, las cuales ya estarán definidas automáticamente al haber definido la función `(==)`, como `dos /= cuatro` o `triple dos /= doble tres`.

Ej.2.

La clase `Ord` se define para poder utilizar los operadores de comparación `(<), (<=), (>), (>=)`. Para poder definir estos operadores, es necesario haber definido previamente la igualdad. Además, habiendo definido una sola de las cuatro operaciones, se pueden definir las otras tres. Se elije `(<=)` como función a definir y se define el resto como sigue:

```
class  (Eq a) => Ord a  where
  (<),(<=),(>),(>=)  :: a -> a -> Bool
  (>) = \ x y -> not (x <= y)
  (>=) = \ x y -> x > y || x == y
  (<)  = \ x y -> not (x >= y)
```

Defina la instancia de `Ord` para `N`:

1) Defina la instancia de `Ord` para `N`:

```
instance Ord N where
  (<=) = ...
```

2) Pruebe las operaciones de la clase con algunos ejemplos: `tres > cinco`, `cuatro <= dos`, `dos >= dos`.

3) Defina las funciones `minimo::N -> N -> N` y `maximo::N -> N -> N` usando las funciones de la instancia de `Ord` para `N`.

4) Defina la función `min3::N -> N -> N -> N` que calcula el mínimo de tres números naturales. Puede hacerse en un renglón, sin utilizar `case`, con las funciones definidas arriba.

Ej.3.
La clase `Num` se define para poder utilizar los símbolos usuales de las operaciones aritméticas:

```
class  (Eq a, Show a) => Num a  where
  (+),(*),(-)  :: a -> a -> a
```

1) Defina la instancia de `Num` para `N`:

```
instance Num N where
  (+) = ...
  (*) = ...
  (-) = ...
```

2) Pruébelas con algunos ejemplos: `tres + cinco`, `cuatro * dos`, `dos * tres - cuatro`.

   Observación: La resta de naturales deberá ser cero si el sustraendo es mayor que el minuendo. Por ejemplo: `tres - cuatro = 0`.

3) Defina las función potencia `(%)::N->N->N`.
   Calcule `tres % dos`, `cuatro % tres`.

4) Reescriba la función `doble` utilizando `(*)`. La definición debe hacerse sin usar `case`, en una sola línea.

5) Defina la función `fact::N->N`, que recibe un natural `n` y calcula su factorial.
   Calcule `fact tres`, `fact (fact tres)`, `fact (fact (fact tres))`.

6) Defina la función `sumi::N->N`, que recibe un natural `n` y calcula la sumatoria de todos los naturales menores o iguales que `n` (o sea, $\sum_{i=0}^{\mathtt{n}} i$).

7) Defina la función `sumdobles::N->N`, que recibe un natural `n` y calcula la sumatoria de los dobles de todos los na turales menores o iguales que `n` (o sea, $\sum_{i=0}^{\mathtt{n}} 2i$).

8) Defina la función `sumfacts::N->N`, que recibe un natural `n` y calcula la sumatoria de los factoriales de todos los naturales menores o iguales que `n` (o sea, $\sum_{i=0}^{\mathtt{n}} i!$).

9) Defina la función `sumfi::(N->N)->N->N`, que recibe un natural `n` y una función `f` y computa la sumatoria de `(f i)` para `i = 0,...n` ($\sum_{i=0}^{\mathtt{n}}(f\ i)$).

10) Reescriba las funciones `sumdobles` y `sumfacts` utilizando `sumfi` (un renglón).

11) Defina la función `sumpares::N->N`, que recibe un natural `n` y computa la sumatoria de todos los naturales menores o iguales que `n` que son pares.

12) Defina la función `sumimpares::N->N`, que recibe un natural `n` y computa la sumatoria de todos los naturales menores o iguales que `n` que son impares.

13) Defina la función `sumpi::(N->Bool)->N->N`, que recibe un natural `n` y un predicado `p` y computa la sumatoria de todos los naturales menores o iguales que `n` para los cuales `p` es verdadero.

14) Reescriba las funciones `sumpares` y `sumimpares` utilizando `sumpi` (un renglón).

15) Defina la función `sumcuadimp::N->N`, que recibe un natural `n` y calcula la suma de los cuadrados de todos los naturales menores o iguales que `n` que son impares.