# Lección 04

## Introducción

En el mundo `Haskell` un tipo es conceptualmente algo muy similar a lo que en matemática conocemos como un álgebra. Se corresponde con un dominio y unas aplicaciones que se le aplican; sabemos que lo hacen porque en el tipo que le corresponde a las mismas aparece el tipo en cuestión.

`Haskell` está diseñado sobre la base de un tipado estático, es decir, en tiempo de compilación cada expresión tiene atribuido un tipo bien determinado que nada en el lenguaje tiene encomendado mutarlo. Un programa que intenta efectuar una labor inapropiada sobre una expresión por el tipo de la misma, encontrará una incongruencia que impedirá la compilación; esto evitará que el programa sea compilado y que sin embargo falle.

El compilador (resp. intérprete) puede razonar sobre el programa vía el tipado antes de compilarlo (resp. interpretarlo), así que no es estrictamente necesario especificar el tipo de las expresiones. No obstante, hacerlo produce un código más legible y de más rápida ejecución. Los tipos más destacados son:

- `Int` es atribuido a números enteros comprendidos entre `-9223372036854775808` y `9223372036854775807` en arquitectura de `64bits`, como podemos comprobar con el siguiente diálogo:

  ```
  Prelude> minBound :: Int
  -9223372036854775808
  Prelude> maxBound :: Int
  9223372036854775807
  ```

  Los detalles pueden ser comprendidos en la documentación de `Data.Int`.

- `Interger` es atribuido a números enteros pero sin cota esta vez. La extensión es a costa de la efectividad, ya que `Int` es más eficiente.

- `Float` es atribuido a números en coma flotante de simple precisión.

- `Double` es atribuido a números en coma flotante de doble precisión.

- `Bool` es el tipo booleano. Toma los valores `True` y `False`.

- `Char` es atribuido a los caractéres. Se define delimitando el carácter por comillas simples. `String` es la abreviatura de lista de caracteres.

- `Ordering` es un tipo que admite tres valores distintos: `GT`, `EQ` y `LT`, que deben ser comprendidos respectivamente como: *mayor que*, *igual que* y *menor que*. La información que podemos tener es la aportada en el siguiente diálogo:

  ```
  Prelude> :i Ordering
  type Ordering :: *
  data Ordering = LT | EQ | GT
    -- Defined in ‘GHC.Types’
  instance Eq Ordering -- Defined in ‘GHC.Classes’
  instance Monoid Ordering -- Defined in ‘GHC.Base’
  instance Ord Ordering -- Defined in ‘GHC.Classes’
  ```

```
  instance Semigroup Ordering -- Defined in ‘GHC.Base’
  instance Enum Ordering -- Defined in ‘GHC.Enum’
  instance Show Ordering -- Defined in ‘GHC.Show’
  instance Bounded Ordering -- Defined in ‘GHC.Enum’
  instance Read Ordering -- Defined in ‘GHC.Read’
```

- `Either` es un tipo incluido en `Data.Either` y del que podemos obtener la siguiente información:

```
  Prelude> :m  + Data.Either
  Prelude Data.Either> :i Either
  type Either :: * -> * -> *
  data Either a b = Left a | Right b
        -- Defined in ‘Data.Either’
  instance Applicative (Either e) -- Defined in ‘Data.Either’
  instance (Eq a, Eq b) => Eq (Either a b)
      -- Defined in ‘Data.Either’
  instance Functor (Either a) -- Defined in ‘Data.Either’
  instance Monad (Either e) -- Defined in ‘Data.Either’
  instance (Ord a, Ord b) => Ord (Either a b)
      -- Defined in ‘Data.Either’
  instance Semigroup (Either a b) -- Defined in ‘Data.Either’
  instance (Show a, Show b) => Show (Either a b)
      -- Defined in ‘Data.Either’
  instance (Read a, Read b) => Read (Either a b)
      -- Defined in ‘Data.Either’
  instance Foldable (Either a) -- Defined in ‘Data.Foldable’
  instance Traversable (Either a) -- Defined in ‘Data.Traversable’
```

- `()` es el tipo de la tupla vacía y sólo tiene el valor `()`. Es también `mempty` de `Monoid`. La información que podemos obtener es la aportada por el siguiente diálogo:

```
  Prelude> :i ()
  type () :: *
  data () = ()
    -- Defined in ‘GHC.Tuple’
  instance Eq () -- Defined in ‘GHC.Classes’
  instance Monoid () -- Defined in ‘GHC.Base’
  instance Ord () -- Defined in ‘GHC.Classes’
  instance Semigroup () -- Defined in ‘GHC.Base’
  instance Enum () -- Defined in ‘GHC.Enum’
  instance Show () -- Defined in ‘GHC.Show’
  instance Bounded () -- Defined in ‘GHC.Enum’
  instance Read () -- Defined in ‘GHC.Read’
```

## Clases

Las clases de tipos simulan una interfaz que regula cierto tipo de comportamiento. Al incluir un tipo en una clase, primitiva o previamente definida, estamos adaptando ese tipo al comportamiento que impone la clase de tipos. La introducción de un tipo en una clase puede ser comprendido como una acto de sobrecarga de operaciones.

Las clases en `Haskell` no son comparables a las homónimas de los leguajes orientados a objetos; son más bien como las interfaces de Java o los protocolos de Objective-C, aunque con mejor comportamiento.

El símbolo `=>` está relacionado con las clases. Es utilizado en la definición de tipos para restringir lo definido a la pertenencia a una ciertas clases, las relacionadas a su izquierda. Por ejemplo:

```
ghci> :t (>)
(>) :: (Ord a) => a -> a -> Bool
```

indica que `>` es una aplicación (operación) infija que actúa sobre instancias del tipo `a` a condición de que `a` esté incluido en la clase `Ord`. La función `elem` tiene el tipo:

```
(Foldable t, Eq a) => a -> t a -> Bool
```

exigiendo que `a` sea un tipo incluido en `Eq`, condición sin la cual no podría actuar `elem`.

Para conocer detalles de una clase basta con usar `:i`. Por ejemplo:

```
Prelude> :i Eq
```

daría extensa información sobre los detalles de la clase `Eq`. Seguidamente damos una breve explicación sobre algunas clases primitivas:

### Clase `Eq`

Los tipos que la integran soportan comparaciones por igualdad. Para la inclusión de un tipo en `Eq` es obligatorio definir para él una de las funciones: `==` o `/=`. Los tipos mencionados más arriba están todos incluidos en `Eq`.

```
Prelude> 5 == 4+1
True
Prelude> 'a' /= 'a'
False
Prelude> 'a' == 'a'
True
Prelude> "Comparar sobre String" == "Comparar sober String"
False
Prelude> 2.718182 == 2.718182
True
```

Definiremos el tipo de dato `RGB` y lo incluiremos en la clase `Eq`:

```
data RGB = Red | Green | Blue

instance Eq RGB where
  Blue  == Blue  = True
  Red   == Red   = True
  Green == Green = True
  _ == _         = False
```

lo que provee el siguiente diálogo:

```
Prelude> :l Prueba04
[1 of 1] Compiling Prueba04         ( Prueba04.hs, interpreted )
Ok, one module loaded.
*Prueba04> Blue == Blue
True
*Prueba04> Blue == Red
False
*Prueba04>
```

### Clase `Show`

Un tipo de dato puede ser presentado si, y sólo si, está incluido en la clase `Show`; en tal caso será presentado como una cadenas de caracteres. La inclusión en la clase `Show` requiere la definición de `show` o bien de `showsPrec`:

```
Prelude> show 0
"0"
Prelude> show 2.718182
"2.718182"
Prelude> show 'a'
"'a'"
Prelude> show "Probando, Probando,..."
"\"Probando, Probando,...\""
Prelude> show False
"False"
```

Incluiremos el tipo `RGB` en la clase `Show`:

```
instance Show RGB where
  show Red = show (255,0,0)
  show Green = show (0,255,0)
  show Blue = show (0,0,255)
```

lo cual proporciona el siguiente diálogo:

```
*Prueba04> Red
(255,0,0)
*Prueba04> Green
(0,255,0)
*Prueba04> Blue
(0,0,255)
```

### Clase `Ord`

La clase `Ord` incluye las funciones de comparación: `<`, `<=`, `>=` y `>`. Están relacionadas con la función `compare`, que toma dos instancias de un mismo tipo incluido en la clase `Ord` y devuelve su relación de ordenación, expresada como una instancia del tipo `Ordering`. Por ejemplo:

```
Prelude> "alpha" < "delta"
True
```

```
Prelude> "alpha" `compare` "delta"
LT
Prelude> 5 <= 2
False
Prelude> compare 5 2
GT
```

La inclusión de un tipo en `Ord` puede ser llevada a cabo definiendo `<` y seguidamente dando una definición estándar de `<=` o `>=`. Es condición necesaria la previa inclusión del dato en `Eq`. Veamos un ejemplo:

```
instance Ord RGB where
  Red < Green = True
  Red < Blue  = True
  Green < Blue = True
  _ < _        = False
  x <= y = (x == y) || (x < y)
```

Es también posible incluir un tipo en la clase `Ord` por la definición de `compare`. En el caso de `RGB` el procedimiento podría ser éste:

```
data RVA = Rojo | Verde | Azul deriving (Read,Eq)

instance Ord RVA where
    compare m n
     | m == n = EQ
     | m == Rojo = LT
     | m == Rojo || m == Verde && n == Azul = LT
     | otherwise = GT
```

En el siguiente ejemplo aprovecharemos que el tipo de dato `Ordering` está en la clase `Monoid` según:

```
mempty = EQ
LT <> _ = LT
GT <> _ = GT
EQ <> x = x
```

(téngase en cuenta que `x <> mempty = x` y también `mempty <> x = x`):

```
data Point = Point Int Int

instance Eq Point where
  Point x1 y1 == Point x2 y2 = (x1==x2 && y1==y2)

instance Ord Point where
    compare (Point x1 x2) (Point y1 y2) = (compare x1 y1) <> (compare x2 y2)
```

### Clase `Read`

Muchos han entendido a `Read` como la clase opuesta a `Show` al ser la encargada de leer una cadena de caractéres y devolver un valor de cierto tipo que es miembro de `Read`. Considere el siguiente diálogo:

```
ghci> read "True" || False
True
ghci> read "8.2" + 3.8
12.0
ghci> read "5" - 2
3
ghci> read "[1,2,3,4]" ++ [3]
[1,2,3,4,3]
```

Sin embargo tenemos lo siguiente:

```
*ghci> read "4"
*** Exception: Prelude.read: no parse
```

En el primer diálogo, al leer hacíamos algo con lo obtenido de la lectura involucrándolo en una expresión; en el último no sabe el intérprete qué hacer con lo leído, la expresión en la que estaba involucrado le permitía inducir un tipo lo que ahora no pasa. Cosa distinta sería que nosotros hiciésemos una indicación explícita y definitiva:

```
*ghci> read "28" :: Int
28
*ghci> read "28" :: Double
28.0
*ghci> (read "28" :: Float) * 4
112.0
*ghci> read "[0,1,2]" :: [Int]
[0,1,2]
*ghci> read "(28,'z')" :: (Int,Char)
(28,'z')
```

Si hubiésemos hecho la definición:

```
data RGB = Red | Green | Blue
         deriving Read
```

podríamos tener el siguiente diálogo:

```
*Prueba04> read "Green" :: RGB
(0,255,0)
```

### Clase `Enum`

Al incluir un tipo en la clase `Enum` sus instancias son numeradas y esta numeración puede ser utilizada posteriormente para las operaciones habituales que la requieren, por ejemplo hacer una lista aritmética. La inclusión en `Enum` requiere definir `toEnum` y `fromEnum`. Consideremos el siguiente ejemplo:

```
data MyDataType = Foo | Bar | Baz
                deriving (Show,Eq)

instance Enum MyDataType where
    toEnum 0 = Foo
```

```
    toEnum 1 = Bar
    toEnum 2 = Baz

    fromEnum Foo = 0
    fromEnum Bar = 1
    fromEnum Baz = 2
```

Aunque una definición menos locuaz, más elegante y más efectiva es la siguiente:

```
import Data.Tuple -- para tener swap que funciona como: swap (x,y) = (y,x)
import Data.Maybe -- para tener fromJust

data MyDataType = Foo | Bar | Baz
            deriving (Show,Eq)

table = [(Foo, 0), (Bar, 1), (Baz, 2)]
instance Enum MyDataType where
    fromEnum = fromJust . flip lookup table
    toEnum = fromJust . flip lookup (map swap table)
```

Ahora podremos calcular el siguiente y predecesor de los elementos que los tengan:

```
*Prueba04> succ Foo
Bar
*Prueba04> succ Bar
Baz
*Prueba04> succ Baz
*** Exception: Maybe.fromJust: Nothing
*Prueba04> pred Baz
Bar
*Prueba04> pred Bar
Foo
*Prueba04> pred Foo
*** Exception: Maybe.fromJust: Nothing
*Prueba04>
```

## Definición de Nuestra Propia Clase

Es posible en Haskell definir una clase propia. En lo que sigue damos el ejemplo del tipo de dato `Rac`, que será introducido en diferentes clases y entre ellas una concebida para llevar a cabo normalizaciones:

```
module Rac (module Rac) where

class TiposConNormalizacion b where
      norm :: b -> b                -- normalización

infix :/
data Rac a = a :/ a

instance (Show a, Integral a) => Show (Rac a) where
  showsPrec _ u = shows x.showChar '/'.shows y
```

```
                where
                 (x :/ y) = norm u

instance (Eq a, Num a) => Eq (Rac a) where
 (x :/ y) == (x' :/ y') = x * y' == x' * y

instance Integral a => TiposConNormalizacion (Rac a) where
  norm (x :/ 0) = error "Racional con denominador 0"
  norm (x :/ y) = ((signum (x * y)) * ((abs x) `div` m)) :/ ((abs y) `div` m)
                where m = gcd x y

instance Integral a => Num (Rac a) where
 (x :/ y) + (x' :/ y') = norm ((x * y' + y * x') :/ (y * y'))
 (x :/ y) - (x' :/ y') = (x :/ y) + (negate x' :/ y)
 (x :/ y) * (x' :/ y') = norm ((x * x') :/ (y * y'))
 abs (x :/ y)          = abs x :/ abs y
 signum (x :/ y)       = signum ( x * y ) :/ 1
 fromInteger i         = (fromInteger i) :/ 1

instance (Integral a, Ord a) => Ord (Rac a) where
 compare  u v
  | (x :/ y) == (x' :/ y') = EQ
  | x * y' <= x' * y       = LT
  | otherwise              = GT
  where (x  :/ y)  = norm u
        (x' :/ y') = norm v
```

## Ejercicios

1. Defina el tipo de dato `Seasons` e inclúyalo en las clases: `Eq`, `Show` y `Ord`.
2. Considere el tipo de dato

   ```
   newtype LexInt = LexInt [Int] -- ints in reverse digit order
   ```

   e inclúyalo manualmente en la clase `Eq` y `Ord`.
3. Defina el tipo abstracto `Conjunto` e inclúyalo en un módulo junto a las operaciones habituales en esa estructura. Su definición debe llevar el tipo de dato `Conjunto` a: `Eq`, `Show`, `Ord` y `TiposConNormalizacion`, al menos.
4. Dar una explicación de la segunda inclusión de `MyDataType` en `Enum`. Seguidamente intruzca el tipo `RGB` en `Enum` con la misma técnica.