# Clase de tipuri
# Laborator 6

**Exercițiul 0.1.** Identify in the Haskell language documentation the definitions of the following types of classes from the standard library: `Show`, `Eq`, `Ord`, `Read`, `Enum`, `Num`.

**Exercițiul 0.2.** Identify in the documentation where the list of types that are part of the type classes listed in the exercise **??** is found.

**Exercițiul 0.3.** Definiți un tip de date `Nat` pentru numere naturale reprezentate în baza 2.
Define a data type `Nat` for natural numbers represented in base 2.
Variants:

1. Varianta 1:

   ```
   data Nat = Cons [Bool] -- list of boolean values
   ```

2. Varianta 2:

   ```
   data Nat = Zero | Double Nat | DoubleAddOne Nat
   ```

   In this encoding, the number 7 would be represented as `DoubleAddOne (DoubleAddOne (DoubleAddOne Zero))`.

3. Variant 3: any other approach ...

**Exercițiul 0.4.** Create instances for `Nat` of the classes `Eq`, `Ord`, `Integral`, and `Num`.

**Exercițiul 0.5.** Define a type `Complex a` for complex numbers whose components are of type `a` (e.g., `Complex Int`, `Complex Float`) and instantiate the class `Num`.

**Exercițiul 0.6.** Define your own class `MyOrd`, similar to `Ord`, and:

1. define `Int` as an instance of the class `MyOrd`
2. define `[a]` an instance of `MyOrd` if `a` is an instance of `MyOrd`
3. implement a sorting algorithm `sort :: MyOrd a => [a] -> [a]`

**Exercițiul 0.7.** Let the following data type be: `data Nat = Zero | Succ Nat`. Using `deriving`, test the functions specific to the `Show`, `Eq`, and `Ord` classes. What do you notice?

**Exercițiul 0.8.** For the data type `data Nat = Zero | Succ Nat`, explicitly define an instance of `Show Nat` that uses the string `"o"` instead of `Zero` and `"s"` instead of `Succ`.

**Exercițiul 0.9.** For the data type `data Nat = Zero | Succ Nat`, explicitly define an instance of `Ord Nat`.

**Exercițiul 0.10.** Create a data type `Arb` that models binary trees whose nodes are labeled with integers. For this data type, customize the function `show` in the `Show` class so that the function associates strings of parentheses and integers to trees. For example, `(2(3()())(4()()))` represents the tree with the root node labeled `2` that has two children labeled `3` and `4`, respectively. Notice that only the parentheses appear for the leaf nodes.

**Exercițiul 0.11.** Below we have a more general definition (than the one in the previous exercise) for binary trees. For this data type, particularize the function `show` in the class `Show` so that it uses parentheses, as in the previous exercise. Explain why we need `a` to be part of the class `Show`.

```
data Arb a = Leaf
           | Node a (Arb a) (Arb a)
instance (Show a) => Show (Arb a) where
...
```

**Exercițiul 0.12.** Consider the data type `data Nat = Zero | Succ Nat`. Fill in the missing part of the code below:

```
instance Eq Nat where
  ...
```

**Exercițiul 0.13.** Consider the data type `data Arb a = Leaf | Node a (Arb a) (Arb a)`. Add the missing code below:

```
instance (Eq a) => Eq (Arb a) where
  ...
```

**Exercițiul 0.14.** Define a type class `Pretty` that includes the function `prettyPrint : a -> String`. Implement this function for the types `Nat` and `Arb a`.

**Exercițiul 0.15.** Define a type class `MyNum` that includes the function `toInt : a -> Int`. Implement this function for the type `Nat`.

**Exercițiul 0.16.** Consider the data type `data Nat = Zero | Succ Nat`. Fill in the missing part of the code below:

```
instance Num Nat where
  ...
```

**Exercițiul 0.17.** Consider the data type `data List a = Nil | Cons a (List a)`. Fill in the missing part of the code below:

```
instance (Eq a) => Eq (List a) where
  ...
```

**Exercițiul 0.18.** Instantiate the `Functor` class with the `List` type.