

# Clase de tipuri

## Laborator 5

**Exercițiul 0.1.** Identificați în documentația limbajului Haskell definițiile următoarelor clase de tipuri din biblioteca standard: `Show`, `Eq`, `Ord`, `Read`, `Enum`, `Num`.

**Exercițiul 0.2.** Identificați în documentație unde se găsește lista tipurilor care fac parte din clasele de tipuri enumerate la Cerința 0.1.

**Exercițiul 0.3.** Definiți un tip de date `Nat` pentru numere naturale reprezentate în baza 2.  
Variante:

1. Varianta 1:

```
data Nat = Cons [Bool] -- lista de valori de tip boolean
```

2. Varianta 2:

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

În această codare, numărul 7 ar fi reprezentat ca `DoubleAddOne (DoubleAddOne (DoubleAddOne Zero))`.

3. Varianta 3: orice altă abordare ...

**Exercițiul 0.4.** Creați instanțe pentru `Nat` ale claselor `Eq`, `Ord`, `Integral` și `Num`.

**Exercițiul 0.5.** Definiți un tip `Complex a` pentru numere complexe ale căror componente sunt de tip `a` (e.g., `Complex Int`, `Complex Float`) și instanțiați clasa `Num`.

**Exercițiul 0.6.** Definiți propria clasă `MyOrd`, similară cu `Ord`, și:

1. definiți `Int` ca instanță a clasei `MyOrd`

2. definiți `[a]` instanță a `MyOrd` dacă `a` este instanță a `MyOrd`

3. implementați un algoritm de sortare `sort :: MyOrd a => [a] -> [a]`

**Exercițiul 0.7.** Fie următorul tip de date: `data Nat = Zero | Succ Nat`. Utilizând `deriving`, testați funcțiile specifice claselor `Show`, `Eq` și `Ord`. Ce observați?

**Exercițiul 0.8.** Pentru tipul de date `data Nat = Zero | Succ Nat`, definiți explicit o instanță `Show Nat` care folosească șirul "o" în loc de `Zero` și "s" în loc de `Succ`.

**Exercițiul 0.9.** Pentru tipul de date `data Nat = Zero | Succ Nat`, definiți explicit o instanță `Ord Nat`.

**Exercițiul 0.10.** Creați un tip de date `Arb` care modelează arborii binari ale căror noduri sunt etichetate cu numere întregi. Pentru acest tip de date particularizați funcția `show` din clasa `Show` astfel încât funcția să asocieze arborilor șiruri de caractere formate din paranteze și numere întregi. Spre exemplu, `(2(3()())(4()()))` reprezintă arborele cu nodul rădăcină etichetat cu 2 care are doi fii etichetați cu 3 și respectiv 4. Observați că pentru nodurile frunză apar doar parantezele.

**Exercițiul 0.11.** Mai jos avem o definiție mai generală (decât cea de la exercițiul anterior) pentru arbori binari. Pentru acest tip de date particularizați funcția `show` din clasa `Show` astfel încât să folosească paranteze, ca la exercițiul anterior. Explicați de ce avem nevoie ca a să facă parte din clasa `Show`.

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

**Exercițiul 0.12.** Fie tipul de date `data Nat = Zero | Succ Nat`. Completați partea care lipsește din codul de mai jos:

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

**Exercițiul 0.13.** Fie tipul de date `data Arb a = Leaf | Node a (Arb a) (Arb a)`. Adăugați codul care lipsește mai jos:

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

**Exercițiul 0.14.** Definiți o clasă de tipuri `Pretty` care include funcția `prettyPrint : a -> String`. Implementați această funcție pentru tipurile `Nat` și `Arb a`.

**Exercițiul 0.15.** Definiți o clasă de tipuri `MyNum` care include funcția `toInt : a -> Int`. Implementați această funcție pentru tipul `Nat`.

**Exercițiul 0.16.** Fie tipul de date `data Nat = Zero | Succ Nat`. Completați partea care lipsește din codul de mai jos:

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

**Exercițiul 0.17.** Fie tipul de date `data List a = Nil | Cons a (List a)`. Completați partea care lipsește din codul de mai jos:

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

**Exercițiul 0.18.** Instanțiați clasa `Functor` cu tipul `List`.