

# Laboratorul 10 - Programare Logică și Funcțională

Seria 36

Mai 2025

## 1 Exercițiul 1

Reluați tipul `LList` din laboratorul anterior, și scrieți instanța de `Foldable`.

## 2 Exercițiul 2

Fie următoarele tipuri de date algebrice, care reprezintă matrici cu linii de lungime diferită.

```
data Line = L [Int]
data Matrix = M [Line]
```

- a. Scrieți o funcție `linesN :: Matrix -> Int -> [Line]` care primește o matrice, un număr întreg `n` și returnează o listă de linii de lungime `n`.

```
> linesN (M [L [1,2,3], L [4,5], L [2,3,6,8], L [8, 5, 3]]) 3
[L [1,2,3], L [8, 5, 3]]
```

- b. Scrieți o funcție `onlyPosElems :: Matrix -> Int -> Bool` care verifică dacă toate liniile de lungime `n` au elemente strict pozitive.

```
> onlyPosElems (M [L [1, 2, 3], L [4, 5], L [2, 3, 6, 8], L [8, 5, 3]]) 3
True
```

```
> onlyPosElems (M [L[1, 2, 3], L [4, 5], L [2, 3, 6, 8], L [8, 5, 3]]) 3
False
```

## 3 Exercițiul 3

Fie următoarele tipuri de date:

```
type Name = String
```

```
data Prop
  = Var Name
  | F
  | T
  | Not Prop
  | Prop :|: Prop
  | Prop :&: Prop
  deriving Eq
infixr 2 :|:
infixr 3 :&:
```

Tipul `Prop` este o reprezentare a formulelor propoziționale. Variabilele precum `p` și `q` pot fi reprezentate ca `Var "p"`, respectiv `Var "q"`. În plus, constantele booleene `F` și `T` reprezintă falsul și adevărul din logică, operatorul unar `Not` este negația, iar operatorii (infix) binari `:|:` și `:&:` reprezintă disjuncția, respectiv conjuncția.

```
p1 :: Prop
p1 = (Var "P" :|: Var "Q") :&: (Var "P" :&: Var "Q")
```

```
p2 :: Prop
p2 = (Var "P" :|: Var "Q") :&: (Not (Var "P") :&: Not (Var "Q"))
```

**Subpunctul a.** Scrieți implementarea formulei  $p_3 := (P \wedge (Q \vee R)) \wedge ((\neg P \vee \neg Q) \wedge (\neg P \vee \neg R))$

```
p3 :: Prop
p3 = undefined
```

**Subpunctul b.** Faceți tipul `Prop` instanță a clasei de tipuri `Show`, înlocuind `Not`, `:|:` și `:&:` cu `,` `|`, `&` și folosind direct numele variabilelor în loc de construcția `Var "nume"`.

```
instance Show Prop where
  show = undefined
```

**Subpunctul c.** Evaluarea expresiilor. Pentru a evalua o expresie, vom considera un *environment* de evaluare, care asociază valori `Bool` variabilelor propoziționale.

```
type Env = [(Name, Bool)]
```

Pentru a obține valoarea asociată unui `Name` în `Env`, utilizați funcția predefinită

```
lookup :: Eq a => a -> [(a, b)] -> Maybe b
```

Pentru a simplifica, mergând pe ipoteza că mereu vom găsi numele căutat, scrieți o variantă a funcției `lookup`, care să nu mai folosească tipul `Maybe`.

```
impureLookup :: Eq a => a -> [(a, b)] -> b
impureLookup = undefined
```

Țineți cont că, pentru a extrage informația `Just` dintr-un `Maybe`, putem utiliza `fromJust`.

```
fromJust :: Maybe a -> a
```

**Exercițiu.** Definiți funcția `eval` cu semnatura:

```
eval :: Prop -> Env -> Bool
eval = undefined
```

Evaluati cele trei formule definite în diferite evaluări.

## 4 Exercițiul 4

Fie următoarele tipuri de date reprezentând expresii și arbori de expresii.

```
data Expr = Const Int
          | Expr :+: Expr
          | Expr :*: Expr
          deriving Eq

data Operation = Add | Mult deriving (Eq, Show)

data Tree = Lf Int
          | Node Operation Tree Tree
          deriving (Eq, Show)
```

- Instanțiați clasa `Show` pentru tipul `Expr`, astfel încât să se afișeze mai simplu expresiile.
- Scrieți o funcție `evalExp :: Expr -> Int` care să evalueze o expresie, determinând valoarea acesteia.
- Scrieți o funcție `evalArb :: Tree -> Int` care evaluează o expresie, determinând valoarea acesteia.
- Scrieți o funcție `expToArb :: Expr -> Tree` care transformă o expresie în arborele corespunzător.