

# Tipuri de date algebrice

## Laborator 3

### 1 Introducere: cum definim noi tipuri?

În Haskell avem posibilitatea să definim noi tipuri de date. Spre exemplu, putem să ne definim propriul tip ce modelează diverse dispozitive mobile pe care le utilizăm:

```
data MobileDevice = Smartphone
                  | Laptop
                  | Tablet
```

**Observația 1.1.** • Cuvântul cheie `data` este utilizat pentru a crea un nou tip;

- Înainte de `=` avem numele noului tip creat: `MobileDevice`;
- După `=` avem definiția *constructorilor de valori* (eng. *value constructors*) care încep cu literă mare și sunt separați de simbolul `|`: `Smartphone`, `Laptop` și `Tablet`.

**Exercițiul 1.2.** Încărcați în `ghci` un fișier care conține definiția tipului `MobileDevice`. Care sunt tipurile pentru `Smartphone`, `Laptop` și `Tablet`?

Tipului `Bool` care este deja definit în Haskell, arată astfel:

```
data Bool = False
          | True
```

Dacă veți încerca să evaluați în `ghci` direct valoarea `True` observați că `ghci` afișează valoarea `True`:

```
cmd> True
True
```

Totuși, pentru o valoare de tipul `MobileDevice` nu avem același comportament:

```
cmd> Laptop
<interactive>:11:1: error:
• No instance for (Show MobileDevice) arising from a use of ‘print’
• In a stmt of an interactive GHCi command: print it
```

Primum această eroare din cauză că `ghci` nu asociază o reprezentare ca șir de caractere a valorii `Laptop`. Pentru a rezolva această problemă, o soluție este să adăugăm `deriving (Show)` la definiția tipului astfel:

```
data MobileDevice = Smartphone
                  | Laptop
                  | Tablet
                  deriving (Show)
```

Despre `deriving (Show)` e important pentru moment să știm doar că permite construirea unor reprezentări ca șir de caractere pentru valorile acestui tip. Vom discuta pe larg această construcție în cursul următor. După reîncărcarea definiției, în `ghci` obținem comportamentul așteptat:

```
cmd> Laptop
Laptop
```

Pentru moment am definit un tip de date nou care are doar trei valori posibile: `Smartphone`, `Laptop` și `Tablet`. În general dorim să putem construi tipuri care au mai multe valori. Spre exemplu, putem avea tablete de diferite dimensiuni. Pentru a realiza acest lucru, putem adăuga noi *câmpuri* constructorilor de valori:

```
data MobileDevice = Smartphone
                  | Laptop
                  | Tablet Int
                  deriving (Show)
```

Observați că pentru constructorul `Tablet` am adăugat un câmp de tip `Int`. Asta ne va permite să construim mai multe valori de tipul `MobileDevice`:

```
cmd> :t (Tablet 12)
(Tablet 12) :: MobileDevice
cmd> :t (Tablet 15)
(Tablet 15) :: MobileDevice
```

Pentru un constructor de tip putem adăuga mai multe câmpuri. Spre exemplu, putem extinde definiția de mai sus astfel încât să atașăm și marca tabletelor:

```
data MobileDevice = Smartphone
                  | Laptop
                  | Tablet Int String
                  deriving (Show)
```

```
cmd> :t (Tablet 15 "Asus")
(Tablet 15 "Asus") :: MobileDevice
cmd> :t (Tablet 15 "Apple")
(Tablet 15 "Apple") :: MobileDevice
```

**Exercițiul 1.3.** Creați un tip de date `Culori` care să conțină câteva culori. Apoi, modificați tipul `MobileDevice` astfel încât să puteți atașa culori pentru fiecare dispozitiv.

Așa cum era de așteptat, putem defini și funcții peste tipurile noi de date. Atunci când dorim să scriem o funcție peste un tip de date creat de noi, vom folosi *pattern matching* peste valorile tipului. Vom explica acest lucru pe un exemplu: să presupunem că dorim să scriem o funcție care va întoarce pentru fiecare dispozitiv o descriere. Considerând prima definiție

pentru `MobileDevice`, funcția va fi implementată astfel:

```
descriere :: MobileDevice -> String
descriere Laptop      = "Acesta este un laptop de culoare roz."
descriere Tablet      = "Aceasta este o tableta mov."
descriere Smartphone  = "Acesta este un telefon mobil."
```

Observați că funcția este definită pentru fiecare constructor în parte. Astfel, atunci când `descriere` se execută, este căutată definiția funcției corespunzătoare argumentului dat. Dacă argumentul este `Smartphone`, atunci ultima linie este cea corespunzătoare.

**Exercițiul 1.4.** Utilizând tipul de date `Culori` scrieți o funcție care afișează culoarea fiecărui dispozitiv.

## 2 Arbori binari. Arbori binari de căutare.

**Exercițiul 2.1.** Definiți un tip de date pentru arbori binari unde nodurile conțin numere întregi.

```
data Arb = Frunza | Nod Integer Arb Arb deriving (Show, Eq)
```

**Exercițiul 2.2.** Scrieți o funcție recursivă `minBST :: Arb -> Integer` care întoarce elementul minim dintr-un arbore binar de căutare.

Funcția poate fi parțială, în sensul în care nu este definită pentru arborele vid.

Indicație: elementul minim este cel mai din stânga.

**Exercițiul 2.3.** Scrieți o funcție recursivă `maxBST :: Arb -> Integer` care întoarce elementul maxim dintr-un arbore binar de căutare.

Funcția poate fi parțială, în sensul în care nu este definită pentru arborele vid.

Indicație: elementul maxim este cel mai din dreapta.

**Exercițiul 2.4.** Scrieți o funcție care verifică dacă un arbore binar este arbore binar de căutare.

```
isBST :: Arb -> Bool
isBST Frunza = True
...
```

Indicație: pentru fiecare subarbore  $T$  al arborelui dat la intrare, funcția testează că nodul rădăcină al lui  $T$  este:

- (A) mai mare decât toate elementele din subarboarele stâng al lui  $T$  și
- (B) mai mic decât toate elementele din subarboarele drept al lui  $T$ .

Presupunând că subarborii lui  $T$  sunt deja cunoscuți ca fiind arbori binari de căutare, testul (A) poate fi implementat simplu folosind `maxBST`, iar testul (B) folosind `minBST`.

**Exercițiul 2.5.** Scrieți o funcție care caută o valoare de tip întreg într-un arbore binar de căutare.

```
search :: Arb -> Integer -> Bool
```

**Exercițiul 2.6.** Scrieți o funcție care inserează o valoare de tip întreg într-un arbore binar de căutare.

```
insert :: Arb -> Integer -> Arb
```

**Exercițiul 2.7.** Scrieți o funcție care șterge (o instanță a) cel mai mare element.

```
removeMax :: Arb -> Arb
```

**Exercițiul 2.8.** Scrieți o funcție care șterge (o instanță) a unei valori de tip întreg dintr-un arbore binar de căutare. Veți folosi probabil `maxBST` și `removeMax` ca funcții ajutătoare.

```
remove :: Arb -> Integer -> Arb
```

**Exercițiul 2.9.** Scrieți funcții care calculează parcurgerea în pre-ordine, in-ordine și post-ordine a arborilor.

```
preOrder :: Arb -> [Integer]  
inOrder  :: Arb -> [Integer]  
postOrder :: Arb -> [Integer]
```

### 3 Expresii booleene

**Exercițiul 3.1.** Definiți un tip de date pentru expresii booleene (variabile boolene, constante true/false, operatori (și, sau, not)).

**Exercițiul 3.2.** Definiți o funcție pentru simplificarea expresiilor booleene.

**Exercițiul 3.3.** Definiți o funcție pentru aducerea unei expresii booleene în forma normală conjunctivă.