

Paradigme de programare

Laboratorul 7

Mihai Nan

Facultatea de Automatica si Calculatoare
Universitatea Politehnica din Bucuresti

Anul universitar 2020-2021

- 1 Construc[Pleaseinsertintopreamble]ia **type**
- 2 Construc[Pleaseinsertintopreamble]ia **data**
- 3 Structuri infinite - Arbore binar

1 Construc[Pleaseinsertintopreamble]ia **type**

2 Construc[Pleaseinsertintopreamble]ia **data**

3 Structuri infinite - Arbore binar

Construcția `type`

- Construcția `type` ne permite definirea unui sinonim de tip, similar cu `typedef` din C.

```
type Point = (Int, Int)
```

```
p :: Point
```

```
p = (2, 3)
```

Observație

Observăm că Haskell nu face distincția între constructorul perechii `(2,3)` și constructorul `Point`, cele două tipuri fiind identice. Singura restricție este aceea că valorile perechii trebuie să fie de tip `Int`.

- Ce se întâmplă cu secvența?

```
p2 :: Point
```

```
p2 = (2.0, 3.0)
```

← Eroare (pt că 2.0 nu este Int)

- 1 Construc[Pleaseinsertintopreamble]ia `type`
- 2 Construc[Pleaseinsertintopreamble]ia `data`
- 3 Structuri infinite - Arbore binar

Construcția **data**

- Construcția **data** permite definirea de noi tipuri de date algebrice, având următoarea formă:

```
data NumeTip = Constructor1 | Constructor2 | ... | ConstructorN
```

Important

Numele tipului (denumit și **constructor de tip**) → poate fi folosit în expresii de tip (ex. `expr :: NumeTip`);

Numele constructorilor (denumiți și **constructori de date**) → pot fi folosiți în definiții (ex. `expr = Constructor1`).

↓ constructor de date
`data PointT = PointC Double Double deriving Show`

↑ numele tipului

```
p :: PointT
```

```
p = PointC 2.5 3.5
```

getX :: Point T → Double
getX (PointC x -) = x

```
> :t PointC
```

```
PointC :: Double -> Double -> PointT
```

Construcția **data** – Exercițiul 1

Se dă tipul de date **Vector**, reprezentând vectori din spațiul \mathbb{R}^3 .
Implementați produsul scalar (dot product) dintre doi vectori.

$$a = (a_1, a_2, a_3) \quad b = (b_1, b_2, b_3)$$

$$a \bullet b = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

$$v_1 = (1.5, 2, 3.5)$$

```
data Vector = V
{ vx :: Double
, vy :: Double
, vz :: Double
} deriving (Show, Eq)
```

$$v_1 :: \text{Vector}$$

$$v_1 = V \ 1.5 \ 2.0 \ 3.5$$

```
dotV :: Vector -> Vector -> Double
```

$$\text{dotV } (V \ vx_1 \ vy_1 \ vz_1) \ (V \ vx_2 \ vy_2 \ vz_2) = vx_1 * vx_2 + vy_1 * vy_2 + vz_1 * vz_2$$

Construcția `data` – Exercițiul 1

Se dă tipul de date `Vector`, reprezentând vectori din spațiul \mathbb{R}^3 .
Implementați produsul scalar (dot product) dintre doi vectori.

$$a \bullet b = a_1 \cdot b_1 + a_2 \cdot b_2 + a_3 \cdot b_3$$

```
data Vector = V
  { vx :: Double
  , vy :: Double
  , vz :: Double
  } deriving (Show, Eq)

> :t vx
vx :: Vector -> Double
> vx (V 1 2 3)
1.0
```

```
dotV :: Vector -> Vector -> Double
```

$\text{dot } V \ v1 \ v2 = vx \ v1 * vx \ v2 + vy \ v1 * vy \ v2 + vz \ v1 * vz \ v2$

Construcția **data** – Exercițiul 2

Definiți un tip de date BST a pentru a implementa un arbore binar de căutare. Implementați funcții pentru a insera o valoare într-un arbore binar de căutare, căutarea unui element într-un arbore binar de căutare dat, o funcție care întoarce lista elementelor din parcurgerea în inordine a arborelui.

tipul câmpului valoare
pt. arborele vid

```
data BST a = UndefinedNode | BSTNil deriving Show
```

↓?
BSTNode a (BST a) (BST a)

Cum definim nodul frunză cu valoarea 5?

node :: BST Int

node = BSTNode 5 BSTNil BSTNil

```
data Tree a = TreeNode a [Tree a] deriving Show
```

(reprezentare pt. arbore cu oricâți copii)

Construcția `data` – Exercițiul 2

Funcție pentru inserarea unei valori într-un arbore binar de căutare

```
data BST a = BSTNode a (BST a) (BST a) | BSTNil deriving Show
insertElem :: (Ord a, Eq a) => BST a -> a -> BST a
```

Construcția `data` – Exercițiul 2

Funcție pentru căutarea unei valori într-un arbore binar de căutare dat

```
data BST a = BSTNode a (BST a) (BST a) | BSTNil deriving Show
findElem :: (Ord a, Eq a) => BST a -> a -> Maybe a
```

Tipul parametrizat *Maybe*

```
data Maybe a = Just a | Nothing deriving (Show, Eq, Ord)
```

- `a` – este o variabilă de tip (exact ca la `BST`);
- constructorul `Just`
 - > `:t Just`
 - `Just :: a -> Maybe a`
- constructorul `Nothing`
 - > `:t Nothing`
 - `Nothing :: Maybe a`

Observație

Această structură ne este utilă atunci când lucrăm cu funcții care pot eșua în a întoarce o valoare utilă.

Construcția `data` – Exercițiul 2

Funcție pentru căutarea unei valori într-un arbore binar de căutare dat

```
data BST a = BSTNode a (BST a) (BST a) | BSTNil deriving Show
findElem :: (Ord a, Eq a) => BST a -> a -> Maybe a
```

Construcția `data` – Exercițiul 2

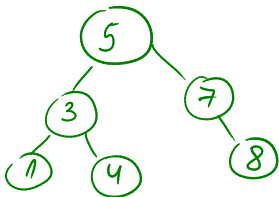
Funcție care întoarce lista elementelor din parcurgerea în inordine a arborelui

```
data BST a = BSTNode a (BST a) (BST a) | BSTNil deriving Show
inorder :: BST a -> [a]
```

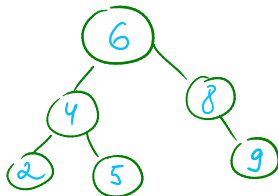
Construcția `data` – Exercițiul 2

Funcționala analogă lui `map`, care să aplice o funcție asupra cheilor nodurilor din arbore

```
data BST a = BSTNode a (BST a) (BST a) | BSTNil deriving Show  
mapTree :: (a -> b) -> BST a -> BST b
```



$$f = \lambda x \rightarrow x + 1$$



Construcția `data` – Exercițiul 2

Funcționala analoagă lui `foldl`, care să parcurgă nodurile în ordinea: rădăcină, `nod_stanga`, `nod_dreapta`

```
data BST a = BSTNode a (BST a) (BST a) | BSTNil deriving Show
foldlTree :: (b -> a -> b) -> b -> BST a -> b
```


f

↑
acc

↑
arbore

↑
rezultatul

Construcția **data** – Exercițiul 2

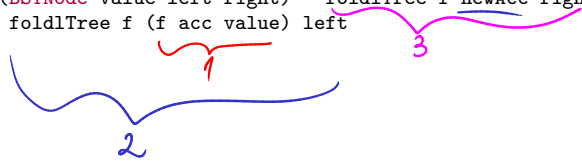
Funcționala analoagă lui **foldl**, care să parcurgă nodurile în ordinea: rădăcină, nod_stanga, nod_dreapta

```
data BST a = BSTNode a (BST a) (BST a) | BSTNil deriving Show  
foldlTree :: (b -> a -> b) -> b -> BST a -> b
```

```
foldlTree _ acc BSTNil = acc
```

```
foldlTree f acc (BSTNode value left right) = foldlTree f newAcc right
```

```
  where newAcc = foldlTree f (f acc value) left
```

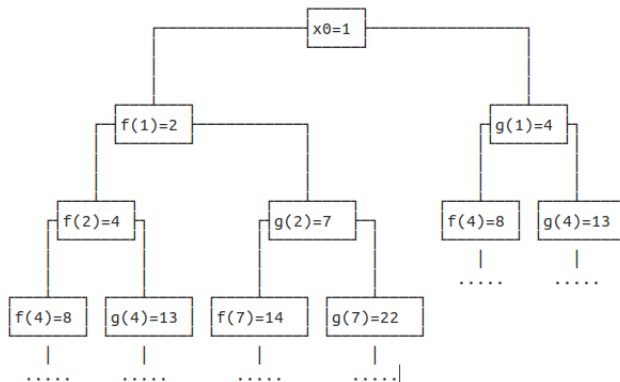


- 1 Construc[Pleaseinsertintopreamble]ia `type`
- 2 Construc[Pleaseinsertintopreamble]ia `data`
- 3 Structuri infinite - Arbore binar

Structuri infinite - Arbore binar

Pornind de la o valoare numerică x_0 , găsiți numărul minim de aplicări de funcții succesive f sau g necesare pentru a ajunge la o valoare target xf .

Fie $f = \backslash x \rightarrow 2 * x$ și $g = \backslash x \rightarrow 3 * x + 1$.



Structuri infinite – Arbore binar

```
data InfBST a = Node
  { value    :: a
  , parent   :: Maybe (InfBST a)
  , func     :: String
  , left     :: InfBST a
  , right    :: InfBST a
  } deriving (Eq, Show)
```

```
f :: (Num a) => a -> a
f = \x -> 2 * x
```

```
g :: (Num a) => a -> a
g = \x -> 3 * x + 1
```

```
completeBinaryTree :: (Show a, Num a) => a -> InfBST a
completeBinaryTree x0 = completeBinaryTreeHelper x0 Nothing ""
```

```
completeBinaryTreeHelper v p fStr = currentNode
  where
    currentNode = Node v p fStr leftNode rightNode
    leftNode    = completeBinaryTreeHelper (f v) (Just currentNode) "f"
    rightNode   = completeBinaryTreeHelper (g v) (Just currentNode) "g"
```