LAB 8

Friday, April 15, 2022 7:54 PM

Î în acest laborator, vom folosi exemple de pe

http://learnyouahaskell.com/making-our-own-types-and-typeclasses#algebraic-data-types

TIPURI SINONIME => punem un alt nume unui tip deja existent

```
type String = [Char]
```

```
type PhoneBook = [(String,String)]
```

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

De obicei, vrem să facem asta doar când folosim tipuri de date lungi sau în contextul nostru, vrem să înțelegem mai bine ce reprezintă ele.

ex: [(String, String)] nu este suficient de expresiv, dar [(Name, PhoneNumber)] da

TIPURI DE DATE:

Până acum am întâlnit mai multe tipuri de date:

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

- Bool = numele tipului
- False | True = value constructors => ne spun ce valori poate avea acest tip
- data = anunță că urmează să definim un tip de date nou

Î în Haskell putem defini și noi tipuri de date:

Cum putem reprezenta o formă geometrică?

ex: un cerc poate fi un tuplu (x, y, R) (x, y = coordonate centru; R = raza)

SAU, am putea să definim propriul nostru tip:

① TIPURI ENUMERATE

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

- Shape = numele tipului
- Circle = value constructor 1
- Rectangle = value constructor 2

Circle = value constr. care primește 3 câmpuri de tipul Float

Rectangle = —"— 4 câmpuri —"—

=> unui value constr. îi putem adăuga opțional niște câmpuri / parametri care ne spun ce tip vrem avea ei

OBS: value constructors sunt funcții care primesc parametrii pe care îi specificăm și întorc o valoare cu tipul de date definit:

```
ghci> :t Circle
Circle :: Float -> Float -> Float -> Shape
ghci> :t Rectangle
Rectangle :: Float -> Float -> Float -> Float -> Shape
```

(Float -> Float -> Float -> Float) — params; (Shape) — res

CUM LE FOLOSIM?

```
1 surface :: Shape -> Float
2 surface (Circle _ _ r) = pi * r ^ 2
3 surface (Rectangle x1 y1 x2 y2) = (abs $ x2 - x1) * (abs $ y2 - y1)
```

1 => tipul funcției este Shape -> Float => funcția primește un param. de tipul Shape și întoarce un Float

Am putea scrie Circle -> Float ? =>

NU, pentru că Circle nu este un tip, ci un constructor. La fel, nu putem defini o funcție cu tipul True -> Float

2, 3 => ce face funcția

PATTERN MATCHING pe constructori

=> pentru fiecare constructor, putem spune ce face funcția

```
ghci> surface $ Circle 10 20 10
314.15927
ghci> surface $ Rectangle 0 0 100 100
10000.0
```

Chiar dacă în acest laborator nu vom vorbi despre type classes, vom folosi 'Show' pentru a putea afișa datele.

↳ ex: Num, Ord, Eq, Show, ...

Dacă încercăm să punem în consolă 'Circle 2.3 3.0 4.2', vedem că primim o eroare. Asta înseamnă că Haskell nu știe să afișeze datele noastre => folosim 'deriving (Show)'

② TIPURI ÎNREGISTRATE

Până acum am văzut cum putem defini noi tipuri de date în acest fel:

```
data Person = Person String String Int Float String String deriving (Show)

guy = Person "Buddy" "Finklestein" 43 184.2 "526-2928" "Chocolate"
```

↳ Când vedem această reprezentare nu prea ne dăm seama ce sunt toți parametrii

```
firstName :: Person -> String
firstName (Person firstname _ _ _ _ _) = firstname

lastName :: Person -> String
lastName (Person _ lastname _ _ _ _) = lastname

age :: Person -> Int
age (Person _ _ age _ _ _) = age

height :: Person -> Float
height (Person _ _ _ height _ _) = height

phoneNumber :: Person -> String
phoneNumber (Person _ _ _ _ number _) = number

flavor :: Person -> String
flavor (Person _ _ _ _ _ flavor) = flavor
```

⇒ funcții care extrag câte un câmp dintr-o persoană

Putem face asta, apelând la înregistrări:

```
data Person2 = Person2 { firstName2 :: String
                       , lastName2 :: String
                       , age2 :: Int
                       , height2 :: Float
                       , phoneNumber2 :: String
                       , flavor2 :: String
                       } deriving (Show)
```

numele câmpului → tipul câmpului

Acum avem funcții care știu să selecteze câmpul dorit.

③ TIPURI PARAMETRIZATE

Până acum am folosit **value constructors**, care primeau param. **valori** și generau un nou tip de date

Similar, putem folosi **type constructors**, care primesc param. **tipuri** și generează un nou tip de date

Exemplu:

```
data Maybe a = Nothing | Just a
```

↳ type parameter

type constructor

Maybe NU este un tip, ci un constructor de tip

(Maybe Int)
(Maybe Char)
(Maybe Person)

} sunt tipuri

③ TIPURI RECURSIVE

Până acum am văzut că un constructor poate primi zero sau mai mulți param.

```
data Shape = Circle Float Float Float | Rectangle Float Float Float Float
```

param.

```
data Maybe a = Nothing | Just a
```

→ param.

Similar, un constructor poate primi ca parametru chiar tipul intors => recursiv

```
data List a = Void | Cons a (List a) deriving Show
```

tipul intors — constructor — param 1 — param 2

[5] => 5 : [] => Cons 5 Void

5 : [] : Cons Void

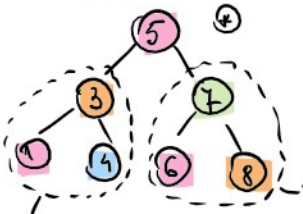
Cons 5 Void : Int : List Int

[5,6] => 5 : 6 : [] => Cons 5 (Cons 6 Empty)

5 : Int

(Cons 6 Empty) : List Int

Binary search tree:



* => fiecare nod are 2 copii → left < nod
right > nod

→ sub arborele drept are numai elem. > 5

sub arborele stg. are numai elem. < 5

=> Un arbore este fie un arbore gol, fie un nod cu o valoare și 2 sub arbori:

```
data BSTree a = EmptyTree | Node a (BSTree a) (BSTree a) deriving (Show)
```

gol — valoarea nodului — sub-arbori

TIPURI SINONIME => punem un alt nume unui tip deja existent

```
type String = [Char]
```

```
type PhoneBook = [(String,String)]
```

```
type PhoneNumber = String
type Name = String
type PhoneBook = [(Name,PhoneNumber)]
```

De obicei, vrem să facem asta doar când folosim tipuri de date lungi sau în contextul nostru, vrem să înțelegem mai bine ce reprezintă ele.

ex: [(String, String)] nu este suficient de expresiv, dar [(Name, PhoneNumber)] da

ex: [[String, String]] nu este suficient de expresiv, dar [(Name, PhoneNumber)] da

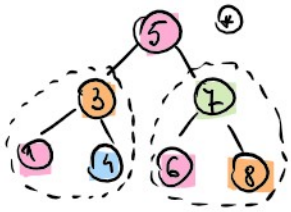
Exerciții

① BST

```haskell
treeInsert :: (Ord a) => a -> BSTree a -> BSTree a
treeInsert x EmptyTree = singleton x
treeInsert x (Node a left right)
    | x == a = Node x left right
    | x < a  = Node a (treeInsert x left) right
    | x > a  = Node a left (treeInsert x right)
```

→ **inorder** => lista de noduri de forma [left , Node , right]
- left ↙ => nodurile din subarborele stg
- right ↳ => nodurile din subarborele drept

=>



=> [1, 3, 4, 5, 6, 7, 8]

→ **balanced** → Un BST balansat este un arbore în care:

- pt fiecare nod => subarborii nu diferă în înălțime cu mai mult de un nivel =>

pt (Node x left right) => |(height left) − (height right)| <= 1

② NESTED LIST

[1, 2, [3, 4], [5, [6, 7]]]
↓
a

```haskell
data NestedList a = Elem a | List [NestedList a]
```

- a ↙ => tipul elementelor
- Elem ↓ => constr 1 => elemente simple
- List ↓ => constr 2 => elemente liste

- emptyList => List []
- consElem 1 2 => [1, 2] => adaugă un elem. într-o Nested_List
  consElem 1 [2,[3,4]] => [1, 2, [3,4]]
- consList [1,2] 3 => [[1,2], 3] => adaugă o listă într-o NestedList
  consList [1,2] [3,4] => [[1,2], 3,4]

- headList 1 => error
  headList [1,[2,3]] => 1
- tailList 1 => error
  tailList [1,[2,3]] => [2,3]

- deepEqual => [1,[2,3]] [1,[2,3]]

List [Elem 1, List [Elem 2, Elem 3]] (l_1) List [Elem 1, List [Elem 2, Elem 3]] (l_2)

=> pt l_1 și l_2 am liste. să verific că toate elem sunt egale => Elem 1 == Elem 1 și List [Elem 2, Elem 3] == List [Elem 2, Elem 3]

=> [Elem 2, Elem 3] == [Elem 2, Elem 3] (l_1', l_2')

Cum aș putea să compar cele 2 liste ?

=> fac o nouă listă în care elem. curent = True, dacă elementele de pe aceeași poz. în l_1' și l_2' sunt egale; False, altfel

=> zipWith deepEqual l_1' l_2'

=> obțin lista [True, True] => rezultatul final o să fie 'and' logic între toate elem. listei rezultat

=>
```
deepEqual (Elem a) (Elem b) = a == b
deepEqual (List l1) (List l2) = and $ zipWith deepEqual l1 l2
deepEqual _ _ = False
```

- flatten [1, [2, [3,4]], 5] => [1,2,3,4,5]

flatten (Elem x) = [x]

flatten (List xs) = ... (*)

Hint: concat [[1,2,3], [4,5]] = [1,2,3,4,5]

↳ primește o listă de liste și le concatenează

List [Elem 1, List [Elem 2, Elem 3]] = List l_1 (l_1)

Cum putem obține [1], [2,3] => map flatten l_1

Cum putem obține [1,2,3] => concat (map flatten l_1)

În Haskell concat + map = concatMap => concatMap flatten xs (*)