Programare funcțională

Introducere în programarea funcțională folosind Haskell

C06

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Tipuri de date algebrice

Tipuri sumă

În Haskell tipul **Bool** este definit astfel:

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

- **Bool** este constructor de tip
- **False** și **True** sunt constructori de date

Tipuri sumă

În Haskell tipul **Bool** este definit astfel:

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

- **Bool** este constructor de tip
- **False** și **True** sunt constructori de date

În mod similar putem defini

```
data Season = Spring | Summer | Autumn | Winter
```

- Season este constructor de tip
- Spring, Summer, Autumn și Winter sunt constructori de date

Tipuri sumă

În Haskell tipul **Bool** este definit astfel:

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

- **Bool** este constructor de tip
- **False** și **True** sunt constructori de date

În mod similar putem defini

```
data Season = Spring | Summer | Autumn | Winter
```

- Season este constructor de tip
- Spring, Summer, Autumn și Winter sunt constructori de date

Bool și Season sunt tipuri de date sumă, adică sunt definite prin **enumerarea alternativelor**.

Tip sumă: Bool

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

Operațiile se definesc prin "pattern matching":

```
not :: Bool -> Bool
not False = True
not True  = False

(&&), (||) :: Bool -> Bool -> Bool
False && q = False
True  && q = q
False || q = q
True  || q = True
```

Tip sumă: Season

```
data Season = Spring | Summer | Autumn | Winter

succesor :: Season -> Season
succesor  Spring = Summer
succesor  Summer = Autumn
succesor  Autumn = Winter
succesor  Winter = Spring
```

Tip sumă: Season

```
data Season = Spring | Summer | Autumn | Winter

succesor :: Season -> Season
succesor  Spring = Summer
succesor  Summer = Autumn
succesor  Autumn = Winter
succesor  Winter = Spring

showSeason :: Season -> String
showSeason Spring = "Primavara"
showSeason Summer = "Vara"
showSeason Autumn = "Toamna"
showSeason Winter = "Iarna"
```

Tipuri produs

Problemă. Să definim un tip de date care să aibă ca valori "punctele" cu două coordonate de tipuri oarecare:

```
data Point a b = Pt a b
```

- Point este constructor de tip
- Pt este constructor de date

Tipuri produs

Problemă. Să definim un tip de date care să aibă ca valori "punctele" cu două coordonate de tipuri oarecare:

```
data Point a b = Pt a b
```

- Point este constructor de tip
- Pt este constructor de date

Pentru a accesa componentele, definim proiecțiile:

```
pr1 :: Point a b -> a
pr1 (Pt x _) = x
pr2 : :Point a b -> b
pr2 (Pt _ y) = y
```

Point este un tip de date produs, definit prin **combinarea** tipurilor a și b.

Tipuri produs

```
data Point a b = Pt a b

Prelude> :t (Pt 1 "c")
(Pt 1 "c") :: Num a => Point a [Char]

Prelude> :t Pt
Pt :: a -> b -> Point a b
-- constructorul de date este operatie

Prelude> :t (Pt 1)
(Pt 1) :: Num a => b -> Point a b
```

Tipuri produs

```
data Point a b = Pt a b
```

```
Prelude> :t (Pt 1 "c")
(Pt 1 "c") :: Num a => Point a [Char]
```

```
Prelude> :t Pt
Pt :: a -> b -> Point a b
-- constructorul de date este operatie
```

```
Prelude> :t (Pt 1)
(Pt 1) :: Num a => b -> Point a b
```

Se pot defini operații:

```
pointFlip :: Point a b -> Point b a
pointFlip (Pt x y) = Pt y x
```

Tipuri de date definite recursiv

Declarația listelor ca tip de date algebric:

```
data List a = Nil
            | Cons a (List a)
```

Tipuri de date definite recursiv

Declarația listelor ca tip de date algebric:

```
data List a = Nil
            | Cons a (List a)
```

- **List** este constructor de tip
- Nil si Cons sunt constructori de date

Tipuri de date definite recursiv

Declarația listelor ca tip de date algebric:

```
data List a  = Nil
             | Cons a (List a)
```

- **List** este constructor de tip
- Nil si Cons sunt constructori de date

Se pot defini operații:

```
append :: List a -> List a -> List a
append Nil ys          = ys
append (Cons x xs) ys = Cons x (append xs ys)
```

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Forma generală:

$$
\begin{aligned}
\mathit{data}\ \mathit{Typename} \quad = \quad & \mathit{Cons}_1\ \ t_{11} \ldots t_{1k_1} \\
& \mid \mathit{Cons}_2\ \ t_{21} \ldots t_{2k_2} \\
& \mid \ldots \\
& \mid \mathit{Cons}_n\ \ t_{n1} \ldots t_{nk_n}
\end{aligned}
$$

unde $k_1, \ldots, k_n \geq 0$

Tipuri de date algebrice

Tipurile de date algebrice se definesc folosind "operațiile" sumă și produs.

Forma generală:

$$\begin{aligned} data\ \ Typename \quad = \quad & Cons_1 \ \ t_{11} \ldots t_{1k_1} \\ & \mid Cons_2 \ \ t_{21} \ldots t_{2k_2} \\ & \mid \ldots \\ & \mid Cons_n \ \ t_{n1} \ldots t_{nk_n} \end{aligned}$$

unde $k_1, \ldots, k_n \geq 0$

- Se pot folosi tipuri sumă și tipuri produs.
- Se pot defini tipuri parametrizate.
- Se pot folosi definiții recursive.

Tipuri de date algebrice

Forma generală:

$$\begin{aligned} \mathit{data}\ \mathit{Typename} \quad = \quad & \mathit{Cons}_1\ \ t_{11} \ldots t_{1k_1} \\ & \mid \mathit{Cons}_2\ \ t_{21} \ldots t_{2k_2} \\ & \mid \ldots \\ & \mid \mathit{Cons}_n\ \ t_{n1} \ldots t_{nk_n} \end{aligned}$$

unde $k_1, \ldots, k_n \geq 0$

Atenție! Alternativele trebuie să conțină **constructori**!

Tipuri de date algebrice

Forma generală:

$$\begin{aligned} \textit{data}\ \ \textit{Typename} \quad = \quad & \textit{Cons}_1\ \ t_{11} \ldots t_{1k_1} \\ & \mid \textit{Cons}_2\ \ t_{21} \ldots t_{2k_2} \\ & \mid \ldots \\ & \mid \textit{Cons}_n\ \ t_{n1} \ldots t_{nk_n} \end{aligned}$$

unde $k_1, \ldots, k_n \geq 0$

Atenție! Alternativele trebuie să conțină **constructori**!

data StrInt = **String** | **Int** este greșit.

Tipuri de date algebrice

Forma generală:

$$\begin{aligned} data\ \ Typename \quad = \quad & Cons_1 \ \ t_{11} \ldots t_{1k_1} \\ & \mid Cons_2 \ \ t_{21} \ldots t_{2k_2} \\ & \mid \ldots \\ & \mid Cons_n \ \ t_{n1} \ldots t_{nk_n} \end{aligned}$$

unde $k_1, \ldots, k_n \geq 0$

Atenție! Alternativele trebuie să conțină **constructori**!

data StrInt = **String** | **Int** este greșit.

data StrInt = VS **String** | VI **Int** este corect.

[VI 1, VS "abc", VI 34, VI 0, VS "xyz"] :: [StrInt]

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

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

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Pair a b = Pair a b
-- constructorul de tip si cel de date pot sa coincida
```

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Pair a b = Pair a b
-- constructorul de tip si cel de date pot sa coincida

data Exp = Lit Int | Add Exp Exp | Mul Exp Exp
```

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Pair a b = Pair a b
-- constructorul de tip si cel de date pot sa coincida

data Exp = Lit Int | Add Exp Exp | Mul Exp Exp

data List a = Nil | Cons a (List a)
```

Tipuri de date algebrice - exemple

```
data Bool = False | True

data Season = Winter | Spring | Summer | Fall

data Shape = Circle Float | Rectangle Float Float

data Pair a b = Pair a b
-- constructorul de tip si cel de date pot sa coincida

data Exp = Lit Int | Add Exp Exp | Mul Exp Exp

data List a = Nil | Cons a (List a)

data Tree a = Empty | Leaf a | Branch (Tree a) (Tree a)
```

Liste cu simboluri

```
data List a = Nil | Cons a (List a)
```

```
data [a] = [] | a : [a]
```

Constructorii listelor sunt [] și : unde

```
[] :: [a]
(:) :: a -> [a] -> [a]
```

Tupluri cu simboluri

```
data (a,b) = (a,b)
data (a,b,c) = (a,b,c)
  ...                 ...
```

Nu există o declarație generică pentru tupluri, fiecare declarație de mai sus definește tuplul de lungimea corespunzătoare, iar constructorii pentru fiecare tip în parte sunt:

```
(,) :: a -> b -> (a,b)
(,,) :: a -> b -> c  -> (a,b,c)
  ...
```

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Declarație ca tip de date algebric folosind șabloane

```
data Nat = Zero | Succ Nat
```

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Declarație ca tip de date algebric folosind șabloane

```
data Nat = Zero | Succ Nat
```

Putem să definim operații, de exemplu:

```
(^^^) :: Float -> Nat -> Float
x ^^^ Zero     = 1.0
x ^^^ (Succ n) = x * (x ^^^ n)
```

Exemplu: numerele naturale (Peano)

Cum definim numerele naturale?

Declarație ca tip de date algebric folosind șabloane

```
data Nat = Zero | Succ Nat
```

Putem să definim operații, de exemplu:

```
(^^^) :: Float -> Nat -> Float
x ^^^ Zero     = 1.0
x ^^^ (Succ n) = x * (x ^^^ n)
```

Comparați cu versiunea folosind notația predefinită:

```
(^^) :: Float -> Int -> Float
x ^^ 0 = 1.0
x ^^ n = x * (x ^^ (n-1))
```

Exemplu: adunare și înmulțire pe Nat

Definiție pe tipul de date algebric:

```
(+++) :: Nat -> Nat -> Nat
m +++ Zero     = m
m +++ (Succ n) = Succ (m +++ n)

(***) :: Nat -> Nat -> Nat
m *** Zero     = Zero
m *** (Succ n) = (m *** n) +++ m
```

Exemplu: adunare și înmulțire pe Nat

Definiție pe tipul de date algebric:

```
(+++) :: Nat -> Nat -> Nat
m +++ Zero     = m
m +++ (Succ n) = Succ (m +++ n)

(***) :: Nat -> Nat -> Nat
m *** Zero     = Zero
m *** (Succ n) = (m *** n) +++ m
```

Comparați cu versiunea folosind notația predefinită:

```
(+) :: Int -> Int -> Int
m + 0 = m
m + n = (m + (n-1)) + 1

(*) :: Int -> Int -> Int
m * 0 = 0
m * n = (m * (n-1)) + m
```

Tipul Maybe (opțiune)

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

Tipul Maybe (opțiune)

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

Rezultate opționale

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)
```

Tipul Maybe (opțiune)

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

Rezultate opționale

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)
```

Argumente opționale

```
power :: Maybe Int -> Int -> Int
power Nothing n    = 2 ^ n
power (Just m) n = m ^ n
```

Maybe - folosirea unui rezultat opțional

```
divide :: Int -> Int -> Maybe Int
divide n 0 = Nothing
divide n m = Just (n `div` m)

-- utilizare gresita
wrong :: Int -> Int -> Int
wrong n m = divide n m + 3

-- utlizare corecta
right :: Int -> Int -> Int
right n m = case divide n m of
                Nothing -> 3
                Just r  -> r + 3
```

Tipul Either (variante)

```
data Either a b = Left a | Right b
```

Tipul Either (variante)

```
data Either a b = Left a | Right b

mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Tipul Either (variante)

```
data Either a b = Left a | Right b

mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care calculează suma elementelor întregi.

```
addints    :: [Either Int String] -> Int
```

Tipul Either (variante)

```
data Either a b = Left a | Right b

mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care calculează suma elementelor întregi.

```
addints    :: [Either Int String] -> Int
addints    []                = 0
addints    (Left n : xs)     = n + addints xs
addints    (Right s : xs) = addints xs
```

Tipul Either (variante)

```
data Either a b = Left a | Right b

mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care calculează suma elementelor întregi.

```
addints    :: [Either Int String] -> Int
addints    []                = 0
addints    (Left n : xs)     = n + addints xs
addints    (Right s : xs) = addints xs

addints' :: [Either Int String] -> Int
addints' xs = sum [n | Left n <- xs]
```

A sau B

```
data Either a b = Left a | Right b

mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care întoarce concatenarea elementelor de tip **String**.

A sau B

```
data Either a b = Left a | Right b

mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care întoarce concatenarea elementelor de tip **String**.

```
addstrs  :: [Either Int String] -> String
addstrs  []             = ""
addstrs  (Left n : xs)  = addstrs xs
addstrs  (Right s : xs) = s ++ addstrs xs
```

A sau B

```
data Either a b = Left a | Right b

mylist :: [Either Int String]
mylist = [Left 4, Left 1, Right "hello", Left 2,
          Right " ", Right "world", Left 17]
```

Definiți o funcție care întoarce concatenarea elementelor de tip **String**.

```
addstrs  :: [Either Int String] -> String
addstrs  []             = ""
addstrs  (Left n : xs)  = addstrs xs
addstrs  (Right s : xs) = s ++ addstrs xs

addstrs' :: [Either Int String] -> String
addstrs' xs = concat [s | Right s <- xs]
```

Utilizarea type

Cu **type** se pot redenumi tipuri deja existente.

Utilizarea type

Cu **type** se pot redenumi tipuri deja existente.

```
type FirstName = String
type LastName  = String
type Age       = Int
type Height    = Float
type Phone     = String

data Person = Person FirstName LastName Age Height Phone
```

Exemplu - date personale. Proiecții

```
data Person = Person FirstName LastName Age Height Phone

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
```

Exemplu - date personale. Utilizare

```
Prelude*> let ionel = Person "Ion" "Ionescu" 20 175.2
      "0712334567"

Prelude*> firstName ionel
"Ion"

Prelude*> height ionel
175.2

Prelude*> phoneNumber ionel
"0712334567"
```

Date personale ca înregistrări

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     }
```

Date personale ca înregistrări

- Putem folosi atât forma algebrică, cât și cea de înregistrare

```
ionel = Person "Ion" "Ionescu" 20 175.2 "
    0712334567"

gigel = Person { firstName = "Gheorghe"
               , lastName="Georgescu"
               , age = 30, height = 192.3
               , phoneNumber = "0798765432"
               }
```

- Putem folosi și pattern-matching
- Proiecțiile sunt definite automat; sintaxă specializată pentru actualizări

```
nextYear :: Person -> Person
nextYear person = person { age = age person + 1 }
```

Date personale ca înregistrări

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     }
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

nextYear person = person { age = age person + 1 }
```

Date personale ca înregistrări

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     }
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

nextYear person = person { age = age person + 1 }
```

```
Prelude> nextYear ionel
```

No instance for (Show Person) arising from a use of ‘print’

Date personale ca înregistrări

```
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     , height :: Float
                     , phoneNumber :: String
                     }
ionel = Person "Ion" "Ionescu" 20 175.2 "0712334567"

nextYear person = person { age = age person + 1 }
```

```
Prelude> nextYear ionel
```

No instance for (Show Person) arising from a use of ‘print’

Deși toate definițiile sunt corecte, o valoare de tip Person nu poate fi afișată deoarece nu este instanță a clasei **Show**.

Quiz time!

Seria 23: https://www.questionpro.com/t/AT4qgZph0s

Seria 24: https://www.questionpro.com/t/AT4NiZphnP

Seria 25: https://www.questionpro.com/t/AT4qgZph1k

Pe săptămâna viitoare!