Programare funcțională

Introducere în programarea funcțională folosind Haskell
C08

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Recapitulare Tipuri de date - Exemple

Maybe

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

Maybe

```
data Maybe a = Nothing | Just a

lookup :: Eq k => k-> [(k,v)] -> Maybe v
lookup k [] = Nothing
lookup k ((k',v):kvs)
  | k' == k   = Just v
  | otherwise = lookup k kvs
```

Liste

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

Liste

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

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

Liste

```
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)
```

Liste

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

list = Cons 2 (Cons 3 (Cons 4 Nil))
```

Liste

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

list = Cons 2 (Cons 3 (Cons 4 Nil))

Prelude> list
Cons 2 (Cons 3 (Cons 4 Nil))
```

Liste

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

instance Show a => Show (List a) where
  show Nil = "[]"
  show (Cons a l) = show a ++ " : " ++ show l

list = Cons 2 (Cons 3 (Cons 4 Nil))
```

Liste

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

instance Show a => Show (List a) where
  show Nil = "[]"
  show (Cons a l) = show a ++ " : " ++ show l

list = Cons 2 (Cons 3 (Cons 4 Nil))

Prelude> list
2 : 3 : 4 : []
```

Liste

```
data  List a  = Vid
  | a ::: List a
  deriving (Show)
infixr 5 :::
```

Liste

```
data  List a  = Vid
  | a ::: List a
  deriving (Show)
infixr 5 :::
```

Exemplu de operație:

```
  (+++) :: List a -> List a -> List a
  infixr 5 +++
  Vid +++ ys          = ys
  (x ::: xs) +++ ys = x ::: (xs +++ ys)
```

Liste

```haskell
data  List2 a  = Vid
  | a ::: List2 a
  deriving Show
infixr 5 :::

list1 = 1 ::: 2 ::: 3 ::: Vid
```

Liste

```
data  List2 a  = Vid
  | a ::: List2 a
  deriving Show
infixr 5 :::

list1 = 1 ::: 2 ::: 3 ::: Vid

Prelunde> list1
1 ::: (2 ::: (3 ::: Vid))
```

Liste

```
data  List2 a  = Vid
                 | a ::: List2 a

infixr 5 :::

instance Show a => Show (List2 a) where
  show Vid = "[]"
  show (a ::: l) = show a ++ " : " ++ show l

list1 = 1 ::: 2 ::: 3 ::: Vid
```

Liste

```
data  List2 a  = Vid
               | a ::: List2 a

infixr 5 :::

instance Show a => Show (List2 a) where
  show Vid = "[]"
  show (a ::: l) = show a ++ " : " ++ show l

list1 = 1 ::: 2 ::: 3 ::: Vid

Prelunde> list1
1 : 2 : 3 : []
```

Expresii Aritmetice

```haskell
data Exp1 = Lit Int | Add Exp1 Exp1 | Mul Exp1 Exp1
  deriving Show

ex1 = Add (Lit 1) (Mul (Lit 5)(Lit 4))
```

Expresii Aritmetice

```
data Exp1 = Lit Int | Add Exp1 Exp1 | Mul Exp1 Exp1
  deriving Show

ex1 = Add (Lit 1) (Mul (Lit 5)(Lit 4))
```

```
Prelude> ex1
Add (Lit 1) (Mul (Lit 5) (Lit 4))
```

Expresii Aritmetice

```
data Exp1 = Lit Int | Add Exp1 Exp1 | Mul Exp1 Exp1
ex1 = Add (Lit 1) (Mul (Lit 5)(Lit 4))
```

Expresii Aritmetice

```haskell
data Exp1 = Lit Int | Add Exp1 Exp1 | Mul Exp1 Exp1
ex1 = Add (Lit 1) (Mul (Lit 5)(Lit 4))

showE :: Exp1 -> String
showE (Lit x) = show x
showE (Add e1 e2) = par e1 ++ " + " ++ par e2
showE (Mul e1 e2) = par e1 ++ " * " ++ par e2

par :: Exp1 -> String
par (Lit x) = showE (Lit x)
par e = "(" ++ showE e ++ ")"

instance Show Exp1 where
  show e = showE e
```

Expresii Aritmetice

```
data Exp1 = Lit Int | Add Exp1 Exp1 | Mul Exp1 Exp1
ex1 = Add (Lit 1) (Mul (Lit 5)(Lit 4))

showE :: Exp1 -> String
showE (Lit x) = show x
showE (Add e1 e2) = par e1 ++ " + " ++ par e2
showE (Mul e1 e2) = par e1 ++ " * " ++ par e2

par :: Exp1 -> String
par (Lit x) = showE (Lit x)
par e = "(" ++ showE e ++ ")"

instance Show Exp1 where
  show e = showE e

Prelude> ex1
1 + (5 * 4)
```

Expresii Aritmetice

```
data Exp2 = L Int | Exp2 :+: Exp2 | Exp2 :*: Exp2
  deriving Show

exp2 = ((L 1) :+: (L 2)) :*: (L 3)
```

Expresii Aritmetice

```
data Exp2 = L Int | Exp2 :+: Exp2 | Exp2 :*: Exp2
  deriving Show

exp2 = ((L 1) :+: (L 2)) :*: (L 3)

Prelude> exp2
(L 1 :+: L 2) :*: L 3
```

Expresii Aritmetice

```
data Exp2 = L Int | Exp2 :+: Exp2 | Exp2 :*: Exp2
exp2 = ((L 1) :+: (L 2)) :*: (L 3)
```

Expresii Aritmetice

```
data Exp2 = L Int | Exp2 :+: Exp2 | Exp2 :*: Exp2
exp2 = ((L 1) :+: (L 2)) :*: (L 3)

showE2 :: Exp2 -> String
showE2 (L x) = show x
showE2 (e1 :+: e2) = par2 e1 ++ " + " ++ par2 e2
showE2 (e1 :*: e2) = par2 e1 ++ " * " ++ par2 e2
par2 :: Exp2 -> String
par2 (L x) = showE2 (L x)
par2 e = "("++ showE2 e ++ ")"

instance Show Exp2 where
  show = showE2
```

Expresii Aritmetice

```
data Exp2 = L Int | Exp2 :+: Exp2 | Exp2 :*: Exp2
exp2 = ((L 1) :+: (L 2)) :*: (L 3)

showE2 :: Exp2 -> String
showE2 (L x) = show x
showE2 (e1 :+: e2) = par2 e1 ++ " + " ++ par2 e2
showE2 (e1 :*: e2) = par2 e1 ++ " * " ++ par2 e2
par2 :: Exp2 -> String
par2 (L x) = showE2 (L x)
par2 e = "("++ showE2 e ++ ")"

instance Show Exp2 where
  show = showE2

Prelude> exp2
(1 + 2) * 3
```

Expresii Logice

```
type Nume = String
data Prop = Var Nume | F | T | Not Prop
          | Prop :|: Prop | Prop :&: Prop
infixr 2 :|:
infixr 3 :&:
```

Expresii Logice

```
type Nume = String
data Prop = Var Nume | F | T | Not Prop
        | Prop :|: Prop | Prop :&: Prop
infixr 2 :|:
infixr 3 :&:

instance Show Prop where
  show (Var nume)= nume
  show (a :|: b) = "("++show a ++ "|" ++ show b++")"
  show (a :&: b) = "("++show a ++ "&" ++ show b++")"
  show (Not p) = "(~"++show p++")"
  show F = "F"
  show T = "T"
```

Expresii Logice

```
type Nume = String
data Prop = Var Nume | F | T | Not Prop
          | Prop :|: Prop | Prop :&: Prop
infixr 2 :|:
infixr 3 :&:

instance Show Prop where
  show (Var nume)= nume
  show (a :|: b) = "("++show a ++ "|" ++ show b++")"
  show (a :&: b) = "("++show a ++ "&" ++ show b++")"
  show (Not p) = "(~"++show p++")"
  show F = "F"
  show T = "T"

Prelude>(Var "P" :|: Var "Q"):&:(Var"P":&:Var "Q")
```

Expresii Logice

```
type Nume = String
data Prop = Var Nume | F | T | Not Prop
          | Prop :|: Prop | Prop :&: Prop
infixr 2 :|:
infixr 3 :&:

instance Show Prop where
  show (Var nume)= nume
  show (a :|: b) = "("++show a ++ "|" ++ show b++")"
  show (a :&: b) = "("++show a ++ "&" ++ show b++")"
  show (Not p) = "(~"++show p++")"
  show F = "F"
  show T = "T"

Prelude>(Var "P":|:Var "Q"):&:(Var"P":&:Var "Q")
((P|Q)&(P&Q))
```

Arbori

```haskell
data Tree = Empty | Leaf Int | Branch Tree Tree
     deriving Show

data Tree2 = Frunza | Nod Int Tree2 Tree2
     deriving Show

tr1 = Branch (Branch (Leaf 2) Empty) (Branch (Branch
   Empty (Leaf 3)) (Leaf 4))

tr2 = Nod 3 (Nod 2 Frunza Frunza ) (Nod 4 (Nod 6
   Frunza Frunza) Frunza)
```

Arbori

```
class ToList a where
  rsd :: a -> [Int]
```

Arbori

```
class ToList a where
  rsd :: a -> [Int]

instance ToList Tree where
  rsd Empty = []
  rsd (Leaf a) = [a]
  rsd (Branch t1 t2) = rsd t1 ++ rsd t2
```

Arbori

```
class ToList a where
  rsd :: a -> [Int]

instance ToList Tree where
  rsd Empty = []
  rsd (Leaf a) = [a]
  rsd (Branch t1 t2) = rsd t1 ++ rsd t2

instance ToList Tree2 where
  rsd Frunza = []
  rsd (Nod r s d) = r : (rsd s) ++ rsd d
```

Arbori

```haskell
tr1 = Branch (Branch (Leaf 2) Empty) (Branch (Branch
    Empty (Leaf 3)) (Leaf 4))

tr2 = Nod 3 (Nod 2 Frunza Frunza ) (Nod 4 (Nod 6
    Frunza Frunza) Frunza)

Prelude> rsd tr1
[2,3,4]

Prelude> rsd tr2
[3,2,4,6]
```

Quiz time!

Seria 23: `https://www.questionpro.com/t/AT4qgZp9GF`

Seria 24: `https://www.questionpro.com/t/AT4NiZp9Eb`

Seria 25: `https://www.questionpro.com/t/AT4qgZp9ju`

Pe săptămâna viitoare!