# Programare declarativă[1]

## Monade

Traian Florin Șerbănuță
Ioana Leuștean

Departamentul de Informatică, FMI, UNIBUC
traian.serbanuta@fmi.unibuc.ro

[1] bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

# Date cu context

# Functor și Applicative

- Functor: **fmap** :: (a -> b) -> m a -> m b

```
Prelude> fmap (*5) (Just 6)
Just 30
Prelude> fmap (*5) [6,6,6]
[30,30,30]
```

# Functor și Applicative

- Functor: **fmap** :: (a -> b) -> m a -> m b

```
Prelude> fmap (*5) (Just 6)
Just 30
Prelude> fmap (*5) [6,6,6]
[30,30,30]
```

- Applicative: **pure** :: a -> m a
  (<*>) :: m(a -> b) -> m a -> m b

```
Prelude> let add2 = (+)
Prelude> let add3 x y z = x+y+z
Prelude> pure add2 <*> Just 1 <*> Just 2
Just 3
Prelude> pure add3 <*> Just 1 <*> Just 2 <*> Just 3
Just 6
```

# Rezolvarea equației de gradul II

## Cu input „parțial”

LA TABLĂ

# Monade

# Clasa de tipuri **Monad**

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
    return = pure

    (>>) :: m a -> m b -> m b
    x >> y = x >>= \_ -> y
```

- m a — tipul computațiilor care produc rezultate de tip a în contextul m
- Tipul a -> m b este tipul continuărilor
- (>>=) este operația de „secvențiere” a computațiilor

# Clasa de tipuri **Monad**

```
class Applicative m => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
    return :: a -> m a
    return = pure

instance Monad Maybe where
        return x = Just x
        Nothing >>= f = Nothing
        Just x >>= f  = f x
```

# Proprietățile monadelor

## Pe scurt

Operația de compunere a continuărilor este asociativă și are element neutru **return**

## Pe mai putin scurt

NeutruD `(return x) >>= g = g x`

NeutruS `x >>= return = x`

Assoc `(fm >>= g) >>= h = fm >>= \ x -> (g x >>= h)`

# **do** notation

```
Prelude> let f x = Just(if x>0 then (sqrt x) else sqrt(- x))
Prelude> let  mx = Just 1
Prelude> mx >>= f
Just 1.0
```

```
mx = Just 1

rez = do
        x <- mx
        return  $ if x>0 then (sqrt x) else sqrt(- x)
```

```
Prelude> rez
Just 1.0
```

# Monad - definiție alternativă

- În orice monadă definim **join** x = x >>= **id**
  Ce face **join**?

# Monad - definiție alternativă

- În orice monadă definim **join** x = x >>= **id**
  Ce face **join**?Este operație de "aplatizare".

```
Prelude Control.Monad> :t join
join :: Monad m => m (m a) -> m a

Prelude Control.Monad> join (Just (Just 1))
Just 1
```

# Monad - definiție alternativă

- În orice monadă definim **join** x = x >>= **id**
  Ce face **join**?Este operație de "aplatizare".

```
Prelude Control.Monad> :t join
join :: Monad m => m (m a) -> m a

Prelude Control.Monad> join (Just (Just 1))
Just 1
```

- Observăm că **fmap**f x = x >>= (**return** . f), deci putem defini **join** și **fmap** în funcție de operațiile monadice (>>=) și **return**.

# Monad - definiție alternativă

- În orice monadă definim **join** x = x >>= **id**
  Ce face **join**?Este operație de "aplatizare".

```
Prelude Control.Monad> :t join
join :: Monad m => m (m a) -> m a

Prelude Control.Monad> join (Just (Just 1))
Just 1
```

- Observăm că **fmap**f x = x >>= (**return** . f), deci putem defini **join** și **fmap** în funcție de operațiile monadice (>>=) și **return**.
- Este posibil și invers: m >>= g = **join** (**fmap** g m)

# Monad - definiție alternativă

- În orice monadă definim **join** x = x >>= **id**
  Ce face **join**?Este operație de "aplatizare".

```
Prelude Control.Monad> :t join
join :: Monad m => m (m a) -> m a

Prelude Control.Monad> join (Just (Just 1))
Just 1
```

- Observăm că **fmap**f x = x >>= (**return** . f), deci putem defini **join** și **fmap** în funcție de operațiile monadice (>>=) și **return**.
- Este posibil și invers: m >>= g = **join** (**fmap** g m)

Așadar monadele pot fi definite folosind: **fmap**, **join**, **return**

# Monada listelor

```
fmap :: (a-> b) -> [a] -> [b]
fmap = map

join :: [[a]]-> [a]
join = concat


instance Monad [] where
        return x = [x]
        xs >>= f = join $ fmap f xs
```

# Efecte laterale, stări, computații nedeterministe

# Logging în C

```
#include <iostream>
#include <sstream>
using namespace std;
ostringstream log;
int increment(int x) {
  log << "Called increment with argument " << x << endl;
  return x + 1;
}

int main() {
  int x = increment(increment(2));
  cout << "Result: " << x << endl << "Log: " << endl << log.str();
}
```

Fiecare apel al lui increment produce un mesaj. Mesajele se acumulează.

# Stare în C

```
#include <iostream>
using namespace std;
int calls;
int increment(int x) {
  calls++;
  return x + calls;
}

int main() {
  int x = increment(increment(2));
  cout << "Result: " << x << endl << "#Calls: " << calls << endl;
}
```

Fiecare apel al lui increment citește starea existentă și o modifică.

# Logging în Haskell

## Funcția originală

```
increment :: Int -> Int
increment x = x + 1
```

## Funcție cu logging

„Îmbogățim” la rezultatul funcției cu mesajul de log.

```
logIncrement :: Int -> (Int, String)
logIncrement x = (x + 1, "Called increment with argument "
    ++ show x ++ "\n")
```

# Logging în Haskell

## Funcția originală

```
increment :: Int -> Int
increment x = x + 1
```

## Funcție cu logging

„Îmbogățim” la rezultatul funcției cu mesajul de log.

```
logIncrement :: Int -> (Int, String)
logIncrement x = (x + 1, "Called increment with argument "
    ++ show x ++ "\n")
```

Problemă: Cum calculăm „logIncrement (logIncrement x)”?

# Stare în Haskell

## Funcția originală în C

```
int increment(int x) {
  return x + calls++;
}
```

## Funcția cu stare în Haskell

Rezultatul este acum o funcție, care dată fiind starea dinaintea executiei, produce un rezultat (folosind eventual starea) și starea cea nouă.

```
type State = Int
stateIncrement :: Int -> (State -> (Int, State))
stateIncrement x = f
  where f calls = (x+calls, calls+1)
```

Problemă: Cum calculăm ”stateIncrement (stateIncrement x)”?

# Computații nedeterministe

## Exemplu folosind grafuri

Un graf orientat este o listă de perechi:
graf = [(1,2),(2,3),(2,7),(1,4),(4,5),(5,6)]

- Pentru fiecare nod funcția succesori întoarce o listă nodurilor care sunt la distanta 1 de el: succesori 1 =[2,4], succesori 2 = [3,7], ...
- Putem aplica acum funcția succesori pentru fiecare element din listă si obținem lista nodurilor care se află la distanță 2 de nod:
pentru nodul 1 se obține lista [3,7,5]

## Computație nedeterministă în Haskell

Rezultatul funcției e listă tuturor valorilor posibile.

```
succesori :: Int -> [Int]
succesori x = [snd p | p <- graf, fst p == x]
```

Problemă: Cum calculăm ”succesori(succesori x)”?

# Cum compunem funcții cu efecte laterale

## Problema generală

Dată fiind funcția `f :: a -> m b` și funcția `g :: b -> m c`, vreau să obțin o funcție `g # f :: a -> m c` care este „compunerea” lui *g* și *f*, propagând efectele laterale.

## Soluție

```
(#) :: (b -> m c) -> (a -> m b) -> (a -> m c)

(g # f) x = f x >>= g
```

## Observație:

Funcția (#) este definită în modulul **Control.Monad**, fiind notată cu (<=<).

# Logging în Haskell

## Funcție cu logging

```
data Log a = Log { val :: a, log :: String }

logIncrement :: Int -> Log Int
logIncrement x = Log (x + 1) ("Called increment with
    argument " ++ show x ++ "\n")

logIncrement2 :: Int -> Log Int
logIncrement2 x = logIncrement x >>= logIncrement
```

# Logging în Haskell

## Funcție cu logging

```
data Log a = Log { val :: a, log :: String }

logIncrement :: Int -> Log Int
logIncrement x = Log (x + 1) ("Called increment with
    argument " ++ show x ++ "\n")

logIncrement2 :: Int -> Log Int
logIncrement2 x = logIncrement x >>= logIncrement
```

## Instanța Monad pentru Log

```
instance Monad Log where
  return a = Log a ""
  ma >>= k = Log {val = val mb, log = log ma ++ log mb}
    where mb = k (val ma)
```

# Stare în Haskell

## Funcția cu stare în Haskell

```
data State state val = St {apply :: state -> (val,state)}

stateIncrement :: Int -> State Int Int
stateIncrement x = St f
  where f calls = (x+calls, calls+1)

stateIncrement2 :: Int -> State Int Int
stateIncrement2 x = stateIncrement x >>= stateIncrement
```

# Stare în Haskell

## Funcția cu stare în Haskell

```
data State state val = St {apply :: state -> (val,state)}

stateIncrement :: Int -> State Int Int
stateIncrement x = St f
  where f calls = (x+calls, calls+1)

stateIncrement2 :: Int -> State Int Int
stateIncrement2 x = stateIncrement x >>= stateIncrement
```

## Instanța Monad pentru stare

```
instance Monad (State state) where
  return a = St (\ s -> (a, s))
  ma >>= k = St g
    where g state = let (val, state') = apply ma state
                    in apply (k val) state'
  -- ma :: State state a, k :: a -> State state b
```

# Computații nedeterministe

## Computație nedeterministă în Haskell

Rezultatul funcției e listă tuturor valorilor posibile.

```
graf = [(1,2),(2,3), (2,7),(1,4),(4,5),(5,6)]
succesori :: Int -> [Int]
succesori x = [snd p | p <- graf, fst p == x]

succesori2 :: Int -> [Int]
succesori2 x = succesori x >>= succesori
```

# Computații nedeterministe

## Computație nedeterministă în Haskell

Rezultatul funcției e listă tuturor valorilor posibile.

```
graf = [(1,2),(2,3), (2,7),(1,4),(4,5),(5,6)]
succesori :: Int -> [Int]
succesori x = [snd p | p <- graf, fst p == x]

succesori2 :: Int -> [Int]
succesori2 x = succesori x >>= succesori
```

## Instanța Monad pentru liste

```
instance Monad [] where
  return a = [a]
  xs >>= k = [y | x <- xs, y <- k x]
```

# Să ne definim propria monadă IO

# Monada MyIO

## partea I

```
module MyIO(MyIO, myPutChar, myGetChar, convert) where
type Input = String
type Output = String


newtype MyIO a =
  MyIO { runMyIO :: Input -> (a, Input, Output) }
```

**Observație: Tipul MyIO is abstract**

- Sunt exportate doar tipul MyIO, myPutChar, myGetChar, convert (și operațiile de monadă)
- Nu este exportat constructorul MyIO și nici operația runMyIO

# Monada MyIO

## partea II

```
myPutChar :: Char -> MyIO ()
myPutChar c = MyIO (\input -> ((), input, [c]))

myGetChar :: MyIO Char
myGetChar = MyIO(\input -> (head input, tail input, ""))
```

### Exemplu

```
runMyIO myGetChar          "abc" == ('a', "bc", "")
runMyIO myGetChar          "bc"  == ('b', "c", "")
runMyIO (myPutChar 'A')    "def" == ((), "def", "A")
runMyIO (myPutChar 'B')    "def" == ((), "def", "B")
```

# Monada MyIO

## partea III

```
instance Monad MyIO where
  return x = MyIO $ \ input -> (x, input, "")
  m >>= k  = MyIO $ \ input ->
    let (x, inputx, outputx) = runMyIO m input
        (y, inputy, outputy) = runMyIO (k x) inputx
     in (y, inputy, outputx ++ outputy)
```

### Exemplu

```
runMyIO
  (do { x <- myGetChar ; y <- myGetChar ; return [x,y] } )
  "abc"
   == ("ab", "c", "")
runMyIO (myPutChar 'A' >> myPutChar 'B') "def"
   == ((), "def", "AB")
runMyIO (myGetChar >>= \x -> myPutChar (toUpper x)) "abc"
   == ((), "bc", "A")
```

# Monada MyIO

## partea IV

```
myInOut m = (\ input ->
                let ((), _, output) = runMyIO m input
                 in output)

convert :: MyIO () -> IO ()
convert m = interact $ concat.(map (myInOut m)).lines
```

Unde

```
interact :: (String -> String) -> IO ()
```

face parte din biblioteca standard, si face următoarele:

- Citește stream-ul de intrare la un șir de caractere (leneș)
- Aplică funcția dată ca parametru acestui șir
- Trimite șirul rezultat către stream-ul de ieșire (tot leneș)

# Folosirea monadei MyIO

## partea I

```
module MyEcho where

import Char
import MyIO

myPutStr :: String -> MyIO ()
myPutStr = foldr (>>) (return ()) . map myPutChar

myPutStrLn :: String -> MyIO ()
myPutStrLn s = myPutStr s >> myPutChar '\n'
```

# Folosirea monadei MyIO

## partea II

```
myGetLine :: MyIO String
myGetLine = myGetChar >>= \x ->
              if x == '\n' then
                return []
              else
                myGetLine >>= \xs ->
                return (x:xs)

myEcho :: MyIO ()
myEcho = myGetLine >>= \line ->
           if line == "" then
             return ()
           else
             myPutStrLn (map toUpper line) >>
             myEcho

main :: IO ()
main = convert myEcho
```

# În execuție

## partea I

```
10-monade$ runghc MyEcho
This is a test.
THIS IS A TEST.
It is only a test.
IT IS ONLY A TEST.
Were this a real emergency, you'd be dead now.
WERE THIS A REAL EMERGENCY, YOU'D BE DEAD NOW.

10-monade$
```

# Folosind notația **do**

```
myGetLine :: MyIO String
myGetLine = do
    x <- myGetChar
    if x == '\n' then
        return []
    else do
        xs <- myGetLine
        return (x:xs)

myEcho :: MyIO ()
myEcho = do
    line <- myGetLine
    if line == "" then
        return ()
    else do
        myPutStrLn (map toUpper line)
        myEcho
```