

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C01

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Organizare

Instructori

- Curs
 - Seria 23: Ana Iova
 - Seria 24: Denisa Diaconescu
 - Seria 25: Ana Iova
- Laborator
 - 231: Ana Iova
 - 232: Natalia Ozunu
 - 233: Miruna Zăvelcă
 - 234: Miruna Zăvelcă
 - 241: Andrei Burdușa
 - 242: Andrei Burdușa
 - 243: Andrei Burdușa
 - 244: Andrei Văcaru
 - 251: Ana Iova
 - 252: Ana Iova

Resurse

- Moodle
- Suporturile de curs si laborator, forumuri, resurse electronice
- Stiri legate de curs vor fi anunțate la curs si posteate pe Moodle
- Link direct catre suporturile de curs si laborator

<https://tinyurl.com/54ftc5wa>

Prezență

Prezența la curs sau la laboratoare nu este obligatorie,
dar extrem de încurajată.

Evaluare

Notare

- Nota finală: 1 (oficiu) + parțial + examen
- Restanță: 1 (oficiu) + examen
(parțialul nu se ia în calcul la restanță)

Condiție de promovabilitate

- cel puțin **5** > 4.99

Examen parțial

- valorează **3 puncte** din nota finală
- durata 45 min
- în săptămâna 7, în cadrul cursului, prima ora de curs
- se va da pe Moodle, online
- nu este obligatoriu și nu se poate reface
- cursul 7 se va tine online pe Teams
- va conține întrebări grilă asemănătoare cu cele din curs
- materiale ajutătoare: suporturile de curs și de laborator

Examen final

- valorează **6 puncte** din nota finală
- durata 1 ora
- în sesiune, fizic
- acoperă toată materia
- va conține exerciții asemănătoare cu cele de la laborator
- materiale ajutătoare: suporturile de curs și de laborator

Puncte bonus

Activitate laborator

- La sugestia instructorilor de la laborator, se poate nota activitatea în plus față de cerințele obișnuite.
- Maxim 1 punct (bonus la nota finală)

Punctele bonus nu se pot folosi în condiția de promovabilitate!

Punctele bonus nu se pot folosi în restanță!

Programare funcțională în Haskell

- Tipuri, functii
- Recursivitate, liste
- Funcții de nivel înalt
- Polimorfism
- Tipuri de date algebrice
- Clase de tipuri
- Functori
- Monade

Resurse suplimentare

- Pagina Haskell
<http://haskell.org>
- Hoogle
<https://www.haskell.org/hoOGLE>
- Haskell Wiki
<http://wiki.haskell.org>
- Cartea online „Learn You a Haskell for Great Good”
<http://learnyouahaskell.com/>

The Haskell Foundation

- <https://haskell.foundation/>
- Fundatie independenta si non-profit ce are ca scop imbunatatirea experientei cu limbajul Haskell
- Ofera suport pentru librarii, tool-uri, educatie, cercetare

Nu trăsați, cereti-ne ajutorul!



De ce programare funcțională?

Programare declarativă vs. imperativă – Ce vs. cum

Programare imperativă (Cum)

Explic mașinii, pas cu pas, algoritmice, **cum** să facă ceva și se întâmplă **ce** vroiam să se întâpte ca rezultat al execuției mașinii.

Programare declarativă vs. imperativă – Ce vs. cum

Programare imperativă (Cum)

Explic mașinii, pas cu pas, algoritmice, **cum** să facă ceva și se întâmplă **ce** vroiam să se întâmple ca rezultat al execuției mașinii.

Programare declarativă (Ce)

Îți spun mașinii **ce** vreau să se întâmple și o las pe ea să se descurce **cum** să realizeze acest lucru. :-)

Programare declarativă vs. imperativă – Ce vs. cum

Programare imperativă (Cum)

Explic mașinii, pas cu pas, algoritmic, **cum** să facă ceva și se întâmplă **ce** vroiam să se întâmple ca rezultat al execuției mașinii.

- limbaje de programare procedurale
- limbaje de programare orientate pe obiecte

Programare declarativă (Ce)

Îți spun mașinii **ce** vreau să se întâmple și o las pe ea să se descurce **cum** să realizeze acest lucru. :-)

Programare declarativă vs. imperativă – Ce vs. cum

Programare imperativă (Cum)

Explic mașinii, pas cu pas, algoritmic, **cum** să facă ceva și se întâmplă **ce** vroiam să se întâmple ca rezultat al execuției mașinii.

- limbaje de programare procedurale
- limbaje de programare orientate pe obiecte

Programare declarativă (Ce)

Îl spun mașinii **ce** vreau să se întâmple și o las pe ea să se descurce **cum** să realizeze acest lucru. :-)

- limbaje de programare logică
- limbaje de interogare a bazelor de date
- limbaje de programare funcțională

Programare funcțională

Programare funcțională în limbajul vostru preferat de programare:

- Java 8, C++11, Python, JavaScript, ...
- Funcții anonte
- Funcții de procesare a fluxurilor de date: filter, map, reduce

Agregarea datelor dintr-o colecție (JS)

C. Boesch, Declarative vs Imperative Programming - Talk.JS

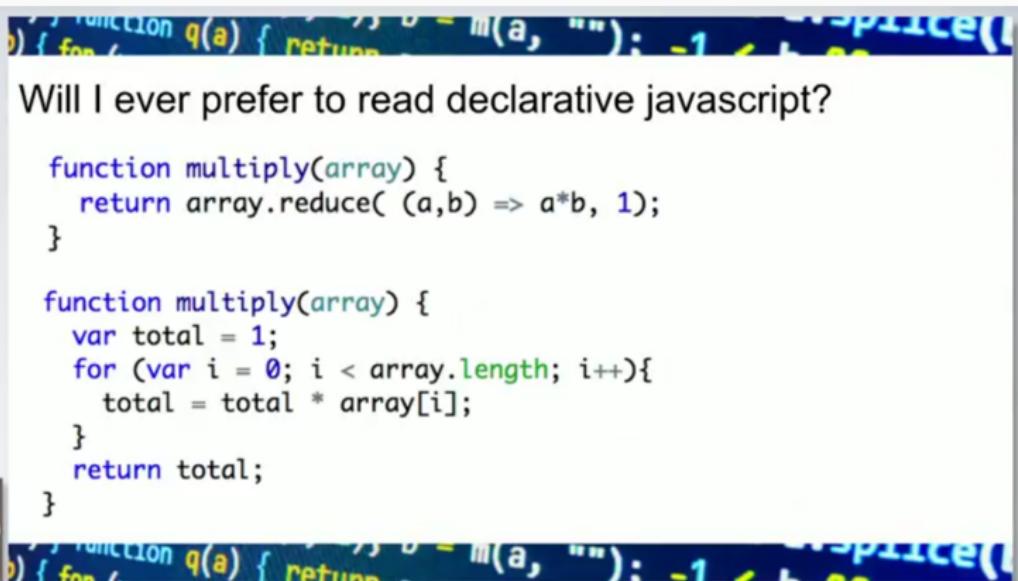
<https://www.youtube.com/watch?v=M2e5sq1rnvc>



Engineers.SG
Meetups Videos

Will I ever prefer to read declarative javascript?

```
function multiply(array) {  
    return array.reduce((a,b) => a*b, 1);  
}  
  
function multiply(array) {  
    var total = 1;  
    for (var i = 0; i < array.length; i++){  
        total = total * array[i];  
    }  
    return total;  
}
```



Agregarea datelor dintr-o colecție (JS)

C. Boesch, Declarative vs Imperative Programming - Talk.JS

<https://www.youtube.com/watch?v=M2e5sq1rnvc>

Reasons to be More Declarative

- Better readability
- Better scalability
- Fewer state-related bugs
- Stand on the shoulders of giants



De ce Haskell? (din cartea Real World Haskell)

The illustration on our cover is of a **Hercules beetle**. These beetles are among the largest in the world. They are also, in proportion to their size, the strongest animals on Earth, able to lift up to 850 times their own weight. Needless to say, we like the association with a creature that has such a high power-to-weight ratio.

Ciurul lui Eratostene

```
primes = sieve [2..]
sieve (p:ps) = p : sieve [ x | x <- ps, mod x p /= 0 ]
```

Haskell este un limbaj funcțional pur



- **Functiile sunt valori.**
- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții
- **Functiile sunt pure:** aceleași rezultate pentru aceleași intrări.
- O bucată de cod nu poate corupe datele altor bucate de cod.
- Distincție clară între părțile pure și cele care comunică cu mediul extern.

Haskell este un limbaj elegant

- Idei abstracte din matematică devin instrumente puternice în practică
 - recursivitate, compunerea de funcții, functori, monade
 - folosirea lor permite scrierea de cod compact și modular
- Rigurozitate: ne forțează să gândim mai mult înainte, dar ne ajută să scriem cod mai corect și mai curat
- Curbă de învățare în trepte
 - Putem scrie programe mici destul de repede
 - Expertiza în Haskell necesită multă gândire și practică
 - Descoperirea unei lumi noi poate fi un drum distractiv și provocator <http://wiki.haskell.org/Humor>

Haskell

- Haskell e lenes (lazy): orice calcul e amânat cât de mult posibil
 - Schimbă modul de concepere al programelor
 - Permite lucrul cu colecții potențial infinite de date precum [1..]
 - Evaluarea lenesă poate fi exploataată pentru a reduce timpul de calcul fără a denatura codul
- firstK k = **take** k primes
- Haskell e minimalist: mai puțin cod, în mai puțin timp, și cu mai puține defecte
 - ... rezolvând totuși problema :-)
- numbers = [1,2,3,4,5]
 total = **foldl** (*) 1 numbers
 doubled = **map** (* 2) numbers
- Oferă suport pentru paralelism și concurență.

Exemplu

```
qsort :: Ord a => [a] -> [a]
qsort []      = []
qsort (p:xs) =
  (qsort lesser) ++ [p] ++ (qsort greater)
where
  lesser  = filter (< p) xs
  greater = filter (>= p) xs
```

Haskell in industrie

Programarea funcțională este din ce în ce mai utilizată în companii.

- Haskell este folosit la Meta, Google, Microsoft, IOHK etc.
- Alte proiecte mari scrise în Haskell:
 - <https://typeable.io/>
 - <https://serokell.io/>
 - <https://xmonad.org/>
 - <https://jaspervdj.be/hakyll/>

Linkuri suplimentare:

[Haskell in industrie - Wiki](#)

[Serokell Blog Post: Best Haskell open source projects](#)

[Typeable Blog Post: 7 Useful Tools Written in Haskell](#)

[Serokell Blog Post: Why Fintech Companies Use Haskell](#)

[Wasp Blog Post: How to get started with Haskell in 2022 \(the straightforward way\)](#)

Haskell in industrie

10 REASONS TO USE HASKELL



MEMORY SAFETY



GARBAGE COLLECTION



NATIVE CODE



STATIC TYPES



RICH TYPES



PURITY



LAZINESS



CONCURRENCY



METAPROGRAMMING



ECOSYSTEM

@serokell

@impurepics

<https://serokell.io/blog/10-reasons-to-use-haskell>

În 1929-1932, A. Church a propus λ -calculul ca sistem formal pentru logica matematică. În 1935 a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată în λ -calcul.

$$\begin{array}{ll} t = & x \quad (\text{variabilă}) \\ & | \lambda x. t \quad (\text{abstractizare}) \\ & | t \ t \quad (\text{aplicare}) \end{array}$$

- În 1935, independent de Church, Turing a dezvoltat mecanismul de calcul numit astăzi **Mașina Turing**.
- În 1936 și el a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing.
- De asemenea, a arătat echivalența celor două modele de calcul.
- Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele, conducând la ceea ce numim astăzi **"Teza Church-Turing"**.

Elemente de bază. Primii pași

Sintaxă

Comentarii

```
-- comentariu pe o linie
{- comentariu pe
   mai multe
   linii -}
```

Identifieri

- siruri formate din litere, cifre, caracterele _ și ' (apostrof)
- identifierii pentru variabile încep cu literă mică sau _
- identifierii pentru tipuri și constructori încep cu literă mare
- Haskell este sensibil la majuscule (case sensitive)

```
double x = 2 * x
data Point a = Pt a a
```

Sintaxă

Blocuri și indentare.

Blocurile sunt delimitate prin indentare.

```
fact n = if n == 0
          then 1
          else n * fact (n-1)
```

```
trei = let
          a = 1
          b = 2
      in a + b
```

Echivalent, putem scrie

```
trei = let {a = 1; b = 2} in a + b
trei = let a = 1; b = 2 in a + b
```

Variabile

Presupunem că fisierul test.hs conține

x=1

x=2

Ce valoare are x?

Variabile

Presupunem că fisierul test.hs conține

x=1

x=2

Ce valoare are x?

Prelude> :l test.hs

```
test.hs:2:1: error:  
  Multiple declarations of 'x'  
    Declared at: test.hs:1:1  
                  test.hs:2:1
```

```
2 | x=2  
 | ^
```

Variabile

În Haskell, variabilele sunt **imutabile** (*immutable*) , adică:

- = **nu** este operator de atribuire
- x = 1 reprezintă o **legatură** (*binding*)
- din momentul în care o variabilă este legată la o valoare, acea valoare nu mai poate fi schimbată

Legarea variabilelor

let .. in ... este o expresie care crează scop local.

Presupunem că fișierul testlet.hs conține

```
x=1  
z= let x=3 in x
```

Legarea variabilelor

let .. in ... este o expresie care crează scop local.

Presupunem că fișierul testlet.hs conține

```
x=1  
z= let x=3 in x
```

```
Prelude> :l testlet.hs  
[1 of 1] Compiling Main  
Ok, 1 module loaded.  
*Main> z  
3  
*Main> x  
1
```

Legarea variabilelor

let .. in ... este o **expresie** care crează scop local.

```
x = let
    z = 5
    g u = z + u
in let
    z = 7
in g 0 + z
```

Ce valoare are x?

Legarea variabilelor

let .. in ... este o **expresie** care crează scop local.

```
x = let
    z = 5
    g u = z + u
in let
    z = 7
in g 0 + z
```

Ce valoare are x?

```
-- x=12
```

Legarea variabilelor

let .. in ... este o **expresie** care crează scop local.

```
x = let
    z = 5
    g u = z + u
in let
    z = 7
in g 0 + z
```

Ce valoare are x?

```
-- x=12
```

```
x = let z = 5; g u = z + u in let z = 7 in g 0
```

Ce valoare are x?

Legarea variabilelor

let .. in ... este o **expresie** care crează scop local.

```
x = let
    z = 5
    g u = z + u
in let
    z = 7
in g 0 + z
```

Ce valoare are x?

```
-- x=12
```

```
x = let z = 5; g u = z + u in let z = 7 in g 0
```

Ce valoare are x?

```
-- x=5
```

Legarea variabilelor

clauza ... **where** ... creaza scop local

f x = g x + g x + z

where

g x = 2*x

z = x-1

Legarea variabilelor

let .. in ... este o expresie

```
x = [ let y = 8 in y , 9]    -- x=[8,9]
```

where este o clauză, disponibilă doar la nivel de definiție

```
x = [y where y = 8, 9] – error: parse error ...
```

Legarea variabilelor

let .. in ... este o expresie

```
x = [let y = 8 in y, 9] -- x=[8,9]
```

where este o clauză, disponibilă doar la nivel de definiție

`x = [y where y = 8, 9] – error: parse error ...`

Variabile pot fi legate și prin *pattern matching* la definirea unei funcții sau expresiei **case**.

```
h x | x == 0      = 0
     | x == 1      = y + 1
     | x == 2      = y * y
     | otherwise   = y
where y = x*x
```

```
f x = case x of
                  0 -> 0
                  1 -> y + 1
                  2 -> y * y
                  _ -> y
where y = x*x
```

Quiz time!

Seria 23 <https://questionpro.com/t/AT4qgZuulH>

Seria 24 <https://tinyurl.com/yc8kdvjm>

Seria 25 <https://questionpro.com/t/AT4qgZuulJ>

Tipuri de date. Sistemul tipurilor

"There are three interesting aspects to types in Haskell: they are strong, they are static, and they can be automatically inferred."

<http://book.realworldhaskell.org/read/types-and-functions.html>

tare – garantează absența anumitor erori

static – tipul fiecărei valori este calculat la compilare

dedus automat – compilatorul deduce automat tipul fiecărei expresii

```
Prelude> :t [( 'a' ,1 , "abc" )]  
[( 'a' ,1 , "abc" )] :: Num b => [ ( Char , b , [ Char ] ) ]
```

Sistemul tipurilor

Tipurile de bază: Int, Integer, Float, Double, Bool, Char, String

Sistemul tipurilor

Tipurile de bază: Int, Integer, Float, Double, Bool, Char, String

Tipuri compuse: tupluri si liste

```
Prelude> :t ('a', True)
('a', True) :: (Char, Bool)
```

```
Prelude> :t ["ana", "ion"]
["ana", "ion"] :: [[Char]]
```

Sistemul tipurilor

Tipurile de bază: Int, Integer, Float, Double, Bool, Char, String

Tipuri compuse: tupluri si liste

```
Prelude> :t ('a', True)
('a', True) :: (Char, Bool)
```

```
Prelude> :t ["ana", "ion"]
["ana", "ion"] :: [[Char]]
```

Tipuri noi definite de utilizator:

```
data RGB = Red | Green | Blue
data Point a = Pt a a      -- tip parametrizat
              -- a este variabila de tip
```

Tipuri de date

Integer: 4, 0, -5

```
Prelude> 4 + 3
```

```
Prelude> (+) 4 3
```

```
Prelude> mod 4 3
```

```
Prelude> 4 `mod` 3
```

Float: 3.14

```
Prelude> truncate 3.14
```

```
Prelude> sqrt 4
```

```
Prelude> let x = 4 :: Int
```

```
Prelude> sqrt (fromIntegral x)
```

Char: 'a','A', '\n'

```
Prelude> import Data.Char
```

```
Prelude Data.Char> chr 65
```

```
Prelude Data.Char> ord 'A'
```

```
Prelude Data.Char> toUpper 'a'
```

```
Prelude Data.Char> digitToInt '4'
```

Tipuri de date

Bool: True, False

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

```
Prelude> True && False || True
```

```
Prelude> not True
```

```
Prelude> 1 /= 2
```

```
Prelude> 1 == 2
```

String: "prog\ndec"

```
type String = [Char] -- sinonim pentru tip
```

```
Prelude> "aa"++"bb"  
"aabb"
```

```
Prelude> "aabb" !! 2  
'b'
```

```
Prelude> lines "prog\ndec"  
[ "prog" , "dec" ]
```

```
Prelude> words "pr og\nde cl"  
[ "pr" , "og" , "de" , "cl" ]
```

Liste

Orice listă poate fi scrisă folosind doar constructorul () și lista vidă [].

- $[1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []$
- $"abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : [])))$
 $\quad == 'a' : 'b' : 'c' : 'd' : []$

Liste

Orice listă poate fi scrisă folosind doar constructorul `(:)` și lista vidă `[]`.

- $[1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []$
- $"abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : [])))$
 $= 'a' : 'b' : 'c' : 'd' : []$

Definiție recursivă. O **listă** este

- **vidă**, notată `[]`, sau
- **compusă**, notată `x:xs`, dintr-un un element `x` numit **capul listei** (*head*) și o listă `xs` numită **coada listei** (*tail*).

Tipuri de date compuse

Tipul listă

```
Prelude>:t [True, False, True]  
[True, False, True] :: [Bool]
```

Tipuri de date compuse

Tipul listă

```
Prelude>:t [True, False, True]  
[True, False, True] :: [Bool]
```

Tipul tuplu – secvențe de tipuri deja existente

```
Prelude> :t (1 :: Int, 'a', "ab")  
(1 :: Int, 'a', "ab") :: (Int, Char, [Char])
```

Prelude> fst (1,'a') -- numai pentru perechi
Prelude> snd (1,'a')

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim dacă introducem comanda?

```
Prelude> :t 1
```

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim dacă introducem comanda?

```
Prelude> :t 1
```

Răspunsul primit este:

```
1 :: Num a => a
```

Semnificația este următoarea:

- Num este o clasă de tipuri
- a este un *parametru de tip*
- 1 este o valoare de tipul a din clasa Num

```
Prelude> :t [1,2,3]
```

```
[1,2,3] :: Num t => [t]
```

Funcții în Haskell. Terminologie

Prototipul funcției

double :: Integer -> Integer

- numele funcției
- signatura funcției

Definiția funcției

double elem = elem + elem

- numele funcției
- parametrul formal
- corpul funcției

Aplicarea funcției

double 5

- numele funcției
- parametrul actual (argumentul)

Exemplu: funcție cu două argumente

Prototipul funcției

add :: Integer -> Integer -> Integer

- numele funcției
- signatura funcției

Definiția funcției

add elem1 elem2 = elem1 + elem2

- numele funcției
- parametrii formali
- corpul funcției

Aplicarea funcției

add 3 7

- numele funcției
- argumentele

Exemplu: funcție cu un argument de tip tuplu

Prototipul funcției

dist :: (Integer, Integer) -> Integer

- numele funcției
- signatura funcției

Definiția funcției

dist (elem1, elem2) = abs (elem1 - elem2)

- numele funcției
- parametrul formal
- corpul funcției

Aplicarea funcției

dist (5, 7)

- numele funcției
- argumentul

Tipuri de funcții

```
Prelude> :t abs  
abs :: Num a => a -> a
```

```
Prelude> :t div  
div :: Integral a => a -> a -> a
```

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

```
Prelude> :t (++)  
(++) :: [a] -> [a] -> [a]
```

```
Prelude> :t zip  
zip :: [a] -> [b] -> [(a, b)]
```

Definirea funcțiilor

```
fact :: Integer -> Integer
```

- Definiție folosind **if**

```
fact n = if n == 0 then 1  
           else n * fact(n-1)
```

- Definiție folosind ecuații

```
fact 0 = 1  
fact n = n * fact(n-1)
```

- Definiție folosind cazuri

```
fact n  
| n == 0      = 1  
| otherwise   = n * fact(n-1)
```

Şabloane (patterns)

```
x:y = [1,2,3] -- x=1 si y =[2,3]
```

Observați că : este constructorul pentru liste.

```
(u,v) = ('a',[ (1,'a'),(2,'b')]) -- u='a',  
-- v=[(1,'a'),(2,'b')]
```

Observați că (,) este constructorul pentru tupluri.

Şabloane (patterns)

Definiții folosind şabloane

```
selectie :: Integer -> String -> String

-- case...of
selectie x s =
    case (x,s) of
        (0,_) -> s
        (1, z:zs) -> zs
        (1, []) -> []
        _ -> (s ++ s)

-- stil ecuational
selectie 0 s = s
selectie 1 (_:s) = s
selectie 1 "" = ""
selectie _ s = s + s
```

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Schimbăm signatura funcției astfel:

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

- are două argumente, de tipuri $(a \rightarrow b)$ și [a],
adică o funcție de la a la b și o listă de elemente de tip a
- întoarce un rezultat de tip [b]

```
Prelude> :t map
```

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

Quiz time!

Seria 23 <https://questionpro.com/t/AT4qgZuulp>

Seria 24 <https://tinyurl.com/5n6s87ks>

Seria 25 <https://questionpro.com/t/AT4qgZuulm>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C02

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Functii

Funcții în Haskell. Terminologie

Prototipul funcției

double :: Integer -> Integer

- numele funcției
- signatura funcției

Definiția funcției

double elem = elem + elem

- numele funcției
- parametrul formal
- corpul funcției

Aplicarea funcției

double 5

- numele funcției
- parametrul actual (argumentul)

Exemplu: funcție cu două argumente

Prototipul funcției

add :: Integer -> Integer -> Integer

- numele funcției
- signatura funcției

Definiția funcției

add elem1 elem2 = elem1 + elem2

- numele funcției
- parametrii formali
- corpul funcției

Aplicarea funcției

add 3 7

- numele funcției
- argumentele

Exemplu: funcție cu un argument de tip tuplu

Prototipul funcției

dist :: (Integer, Integer) -> Integer

- numele funcției
- signatura funcției

Definiția funcției

dist (elem1, elem2) = abs (elem1 - elem2)

- numele funcției
- parametrul formal
- corpul funcției

Aplicarea funcției

dist (5, 7)

- numele funcției
- argumentul

Definirea funcțiilor

```
fact :: Integer -> Integer
```

- Definiție folosind **if**

```
fact n = if n == 0 then 1  
          else n * fact(n-1)
```

- Definiție folosind ecuații

```
fact 0 = 1  
fact n = n * fact(n-1)
```

- Definiție folosind cazuri

```
fact n  
| n == 0      = 1  
| otherwise   = n * fact(n-1)
```

Definirea funcților folosind șabloane și ecuații

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact(n-1)
```

- variabilele și valorile din partea stângă a semnului = sunt șabloane
- când funcția este apelată se încearcă potrivirea parametrilor actuali cu șabloanele, ecuațiile fiind încercate *în ordinea scrierii*
- în definiția factorialului, 0 și n sunt șabloane: 0 se va potrivi numai cu el însuși, iar n se va potrivi cu orice valoare de tip Integer

Definirea funcților folosind şabloane și ecuații

In Haskell, ordinea ecuațiilor este importantă.

Să presupunem că schimbăm ordinea ecuațiilor din definiția factorialului:

```
fact :: Integer -> Integer  
fact n = n * fact(n-1)  
fact 0 = 1
```

Ce se întâmplă?

Definirea funcților folosind şabloane și ecuații

In Haskell, ordinea ecuațiilor este importantă.

Să presupunem că schimbăm ordinea ecuațiilor din definiția factorialului:

```
fact :: Integer -> Integer  
fact n = n * fact(n-1)  
fact 0 = 1
```

Ce se întâmplă?

Deoarece `n` este un pattern care se potrivește cu orice valoare, inclusiv cu 0, orice apel al funcției va alege prima ecuație. Astfel, funcția nu își va încheia execuția.

Definirea funcțiilor folosind şabloane și ecuații

Tipul Bool este definit în Haskell astfel:

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

Putem defini operația || astfel

```
(||) :: Bool -> Bool -> Bool
```

```
False || x = x  
True  || _ = True
```

În acest exemplu şabloanele sunt `_`, **True** și **False**.

Observăm că **True** și **False** sunt constructori de date și se vor potrivi numai cu ei însăși.

Şablonul `_` se numește *wild-card pattern*; el se potrivește cu orice valoare.

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Schimbăm signatura funcției astfel:

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

- are două argumente, de tipuri $(a \rightarrow b)$ și [a],
adică o funcție de la a la b și o listă de elemente de tip a
- întoarce un rezultat de tip [b]

Tipuri de funcții

Fie foo o funcție cu următorul tip

```
foo :: a -> b -> [a] -> [b]
```

- are trei argumente, de tipuri a, b și [a]
- întoarce un rezultat de tip [b]

Schimbăm signatura funcției astfel:

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

- are două argumente, de tipuri $(a \rightarrow b)$ și [a],
adică o funcție de la a la b și o listă de elemente de tip a
- întoarce un rezultat de tip [b]

```
Prelude> :t map
```

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

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZu3yb>

Seria 24: <https://questionpro.com/t/AT4NiZu3J8>

Seria 25: <https://questionpro.com/t/AT4qgZu3yd>

Liste

Liste (slide din C01)

Orice listă poate fi scrisă folosind doar constructorul `(:)` și lista vidă `[]`.

- $[1,2,3] == 1 : (2 : (3 : [])) == 1 : 2 : 3 : []$
- $"abcd" == ['a','b','c','d'] == 'a' : ('b' : ('c' : ('d' : [])))$
 $\quad == 'a' : 'b' : 'c' : 'd' : []$

Definiție recursivă. O **listă** este

- **vidă**, notată `[]`; sau
- **compusă**, notată `x:xs`, dintr-un un element `x` numit **capul listei** (*head*) și o listă `xs` numită **coada listei** (*tail*).

Definirea listelor. Operații

Intervale și progresii

```
interval = ['c'..'e']           -- ['c', 'd', 'e']
progresie = [20,17..1]          -- [20,17,14,11,8,5,2]
progresie' = [2.0,2.5..4.0]     -- [2.0,2.5,3.0,3.5,4.0]
```

Operații

```
Prelude> [1,2,3] !! 2
```

```
3
```

```
Prelude> "abcd" !! 0
```

```
'a'
```

```
Prelude> [1,2] ++ [3]
```

```
[1,2,3]
```

```
Prelude> import Data.List
```

Definiția prin selecție $\{x \mid P(x)\}$

$[E(x) \mid x \leftarrow [x_1, \dots, x_n], P(x)]$

Prelude> xs = [0..10]

Prelude> [x | x <- xs, even x]

[0,2,4,6,8,10]

Prelude> xs = [0..6]

Prelude> [(x,y) | x <- xs, y <- xs, x + y == 10]

[(4,6),(5,5),(6,4)]

Definiția prin selecție $\{x \mid P(x)\}$

$[E(x) \mid x <- [x_1, \dots, x_n], P(x)]$

Putem folosi **let** pentru declarații locale.

```
Prelude> [(i, j) | i <- [1..2],  
                  let k = 2 * i, j <- [1..k]]  
[(1,1),(1,2),  
(2,1),(2,2),(2,3),(2,4)]
```

```
zip xs ys
```

```
Prelude> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

```
Prelude> ys = ['A'.. 'E']
Prelude> zip [1..] ys
[(1, 'A'), (2, 'B'), (3, 'C'), (4, 'D'), (5, 'E')]
```

```
Prelude> xs = ['A'.. 'Z']
Prelude> [x | (i, x) <- [1..] `zip` xs, even i]
"BDFHJLNPRTVXZ"
```

```
zip xs ys
```

Observați diferența!

```
Prelude> zip [1..3] ['A'..'D']
[(1 , 'A') ,(2 , 'B') ,(3 , 'C') ]
```

```
Prelude> [(x,y) | x <- [1..3], y <- ['A'..'D']]
[(1 , 'A') ,(1 , 'B') ,(1 , 'C') ,(1 , 'D') ,
 (2 , 'A') ,(2 , 'B') ,(2 , 'C') ,(2 , 'D') ,
 (3 , 'A') ,(3 , 'B') ,(3 , 'C') ,(3 , 'D') ]
```

Lenevire (Lazyness)

Argumentele sunt evaluate doar când e necesar și doar cât e necesar.

```
Prelude> head []
```

```
*** Exception: Prelude.head: empty list
```

```
Prelude> x = head []
```

```
Prelude> f a = 5
```

```
Prelude> f x
```

```
5
```

```
Prelude> [1,head [],3] !! 0
```

```
1
```

```
Prelude> [1, head [],3] !! 1
```

```
*** Exception: Prelude.head: empty list
```

Liste infinite

Drept consecință a **evaluării lenșe**, se pot defini liste infinite (fluxuri de date)

```
Prelude> natural = [0..]
```

```
Prelude> take 5 natural
```

```
[0,1,2,3,4]
```

```
Prelude> evenNat = [0,2..] -- progresie infinita
```

```
Prelude> take 7 evenNat
```

```
[0,2,4,6,8,10,12]
```

```
Prelude> ones = [1,1..]
```

```
Prelude> zeros = [0,0..]
```

```
Prelude> both = zip ones zeros
```

```
Prelude> take 5 both
```

```
[(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Şabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

[1 ,2 ,3] == 1:[2 ,3] == 1:2:[3] == 1:2:3:[]

Prelude> x:y = [1 ,2 ,3]

Prelude> x

1

Prelude> y

[2 ,3]

Ce s-a întâmplat?

- **x : y** este un şablon pentru liste
- potrivirea dintre **x : y** şi **[1,2,3]** a avut ca efect:
 - "deconstrucţia" valorii [1,2,3] în 1 : [2,3]
 - legarea lui x la 1 şi a lui y la [2,3]

Şabloane (patterns) pentru liste

Definiţii folosind şabloane

```
reverse [] = []
```

```
reverse (x:xs) = (reverse xs) ++ [x]
```

x : xs se potriveşte cu liste nevide

Atenţie!

Şabloanele sunt definite folosind constructori.

De exemplu, operaţia de concatenare pe liste este

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

dar **[x] ++ [1] = [2,1]** **nu** va avea ca efect legarea lui x la 2.

```
Prelude> [x] ++ [1] = [2,1]
```

```
Prelude> x
```

```
error: ...
```

Şabloanele sunt liniare

Sabloanele sunt **liniare**, adică o variabilă apare cel mult odată.

Şabloane în care o variabilă apare de mai multe ori provoacă mesaje de eroare. De exemplu:

```
x:x:[1] = [2,2,1]
```

```
tail (x:x:t) = t
```

```
foo x x = x^2
```

error: Conflicting definitions for x

O soluție este folosirea gărzilor:

```
tail (x:y:t) | (x==y) = t  
| otherwise = ...
```

```
foo x y | (x == y) = x^2  
| otherwise = ...
```

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZu3yi>

Seria 24: <https://questionpro.com/t/AT4NiZu3J9>

Seria 25: <https://questionpro.com/t/AT4qgZu3yp>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C03

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Anunt - examen parțial

- valorează **3 puncte** din nota finală
- durata ~~45 min~~ 40 min
- în săptămâna 7, **in cadrul cursului**
- **test grila pe foaie**
- nu este obligatoriu și nu se poate reface
- ~~cursul 7 se va tine online pe Teams~~
- va conține întrebări grilă asemănătoare cu cele din curs
- materiale ajutătoare: suporturile de curs și de laborator

Operatori. Secțiuni

Operatorii sunt funcții cu două argumente

Operatorii în Haskell

- au două argumente
- pot fi apelați folosind notația infix
- pot fi definiți folosind numai "simboluri" (ex: `*!*`)
 - în definiția tipului operatorul este scris între paranteze
- Operatori predefiniți

`(||) :: Bool -> Bool -> Bool`

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

`(+) :: Num a => a -> a -> a`

- Operatori definiți de utilizator

`(&&&) :: Bool -> Bool -> Bool -- atentie la paranteze`

`True &&& b = b`

`False &&& _ = False`

Functii ca operatori

Operatorii care sunt definiți în formă infix, sunt apelați în formă prefix folosind paranteze

```
2 + 3 == (+) 2 3
```

Operatorii care sunt definiți în formă prefix, sunt apelați în formă infix folosind `` (*backtick*)

```
mod 5 2 == 5 `mod` 2
```

```
Prelude> mod 5 2
```

```
1
```

```
Prelude> 5 `mod` 2
```

```
1
```

```
elem :: a -> [a] -> Bool
```

```
Prelude> 1 `elem` [1,2,3]
```

```
True
```

Precedență și asociativitate

```
Prelude> 3+5*4:[6]++8-2+3:[2]==[23,6,9,2]||True==False  
True
```

Precedence	Left associative	Non-associative	Right associative
9	!!		.
8			^, ^^, **
7	*, /, `div`, `mod`, `rem`, `quot`		
6	+, -		
5			: , ++
4		==, /=, <, <=, >, >=, `elem`, `notElem`	
3			&&
2			
1	>>, >>=		
0			\$, \$!, `seq`

Asociativitate

Operatorul - asociativ la stânga

$$5 - 2 - 1 == (5 - 2) - 1 \quad --- /= 5 - (2 - 1)$$

Operatorul : asociativ la dreapta

$$5 : 2 : [] == 5 : (2 : [])$$

Operatorul ++ asociativ la dreapta

$$(\text{++}) :: [\text{a}] \rightarrow [\text{a}] \rightarrow [\text{a}]$$

$$[] \text{ ++ } \text{ys} = \text{ys}$$

$$(\text{x}:\text{xs}) \text{ ++ } \text{ys} = \text{x}:(\text{xs} \text{ ++ } \text{ys})$$

$$[\text{l}1 \text{ ++ } \text{l}2 \text{ ++ } \text{l}3 \text{ ++ } \text{l}4 \text{ ++ } \text{l}5] == [\text{l}1 \text{ ++ } (\text{l}2 \text{ ++ } (\text{l}3 \text{ ++ } (\text{l}4 \text{ ++ } \text{l}5)))]$$

Secțiuni (operator sections)

Secțiunile operatorului binar (**op**) sunt (**op e**) și (**e op**).

Secțiunile lui (++) sunt (++ e) și (e ++)

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
```

```
Prelude> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
```

```
Prelude> (++ " world!") "Hello"
"Hello world!"
```

```
Prelude> ++ " world!" "Hello"
error
```

Secțiuni (operator sections)

Secțiunile operatorului binar (op) sunt (op e) și (e op).

Secțiunile lui (\leftrightarrow) sunt ($\leftrightarrow e$) și ($e \leftrightarrow$)

Prelude> $x \leftrightarrow y = x - y + 1$ -- definit de noi

Prelude> :t ($\leftrightarrow 3$)
($\leftrightarrow 3$) :: **Num** a => a -> a

Prelude> ($\leftrightarrow 3$) 4
2

Secțiuni

Secțiunile sunt afectate de **asociativitatea și precedența** operatorilor.

```
Prelude> :t (+ 3 * 4)
(+ 3 * 4) :: Num a => a -> a
```

```
Prelude> :t (* 3 + 4)
error -- + are precedenta mai mica decat *
```

```
Prelude> :t (* 3 * 4)
error -- * este asociativa la stanga
```

```
Prelude> :t (3 * 4 *)
(3 * 4 *) :: Num a => a -> a
```

Functii de nivel înalt

Funcții anonime

Funcțiile sunt valori (*first-class citizens*).

Funcțiile pot fi folosite ca argumente pentru alte funcții.

Funcții anonime = lambda expresii

$\lambda x_1 x_2 \dots x_n \rightarrow \text{expresie}$

Prelude> $(\lambda x \rightarrow x + 1) 3$

4

Prelude> inc = $\lambda x \rightarrow x + 1$

Prelude> add = $\lambda x y \rightarrow x + y$

Prelude> aplic = $\lambda f x \rightarrow f x$

Prelude> map $(\lambda x \rightarrow x+1) [1,2,3,4]$

[2,3,4,5]

Functiile sunt valori

Exemplu:

flip :: (a → b → c) → (b → a → c)

- definiția cu lambda expresii

flip f = \x y → f y x

- definiția folosind şabloane

flip f x y = f y x

- **flip** ca valoare de tip funcție

flip = \f x y → f y x

Componerea funcțiilor — operatorul .

Matematic. Date fiind $f : A \rightarrow B$ și $g : B \rightarrow C$, componerea lor, notată $g \circ f : A \rightarrow C$, este dată de formula

$$(g \circ f)(x) = g(f(x))$$

În Haskell.

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)  
(g . f) x = g (f x)
```

Exemplu

```
Prelude> :t reverse
```

```
reverse :: [a] -> [a]
```

```
Prelude> :t take
```

```
take :: Int -> [a] -> [a]
```

```
Prelude> :t take 5 . reverse
```

```
take 5 . reverse :: [a] -> [a]
```

```
Prelude> (take 5 . reverse) [1..10]
```

```
[10,9,8,7,6]
```

```
Prelude> (head . reverse . take 5) [1..10]
```

```
5
```

Operatorul \$

Operatorul (\$) are precedență 0.

$$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$$
$$f \$ x = f x$$

```
Prelude> (head . reverse . take 5) [1..10]  
5
```

```
Prelude> head . reverse . take 5 \$ [1..10]  
5
```

Operatorul (\$) este asociativ la dreapta.

```
Prelude> head \$ reverse \$ take 5 \$ [1..10]  
5
```

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZu3zB>

Seria 24: <https://questionpro.com/t/AT4NiZu3KF>

Seria 25: <https://questionpro.com/t/AT4qgZu3zE>

Procesarea fluxurilor de date: Map, Filter, Fold

**Transformarea fiecărui element dintr-o listă -
map**

Exemplu - Pătrate

Definiți o funcție care pentru o listă de numere întregi dată ridică la pătrat fiecare element din listă.

Soluție descriptivă

```
squares :: [Int] -> [Int]
squares xs = [ x * x | x <- xs ]
```

Soluție recursivă

```
squares :: [Int] -> [Int]
squares []      = []
squares (x:xs) = x*x : squares xs
```

```
Prelude> squares [1,-2,3]
[1,4,9]
```

Exemplu - Coduri ASCII

Transformați un sir de caractere în lista codurilor ASCII ale caracterelor.

Soluție descriptivă

```
ords :: [Char] -> [Int]
ords xs = [ ord x | x <- xs ]
```

Soluție recursivă

```
ords :: [Char] -> [Int]
ords []      = []
ords (x:xs) = ord x : ords xs
```

```
Prelude> ords "a2c3"
[97,50,99,51]
```

Funcția map

Date fiind o funcție de transformare și o listă,
aplicați funcția fiecărui element al unei liste date.

Soluție descriptivă

```
map :: (a -> b) -> [a] -> [b]
map f xs = [ f x | x <- xs ]
```

Soluție recursivă

```
map :: (a -> b) -> [a] -> [b]
map f []      = []
map f (x:xs) = f x : map f xs
```

Exemplu — Pătrate

Soluție descriptivă

```
squares :: [Int] -> [Int]
squares xs = [ x * x | x <- xs ]
```

Soluție recursivă

```
squares :: [Int] -> [Int]
squares []      = []
squares (x:xs) = x*x : squares xs
```

Soluție folosind map

```
squares :: [Int] -> [Int]
squares xs = map sqr xs
            where sqr x = x * x
```

Exemplu — Coduri ASCII

Soluție descriptivă

```
ords :: [Char] -> [Int]
ords xs = [ ord x | x <- xs ]
```

Soluție recursivă

```
ords :: [Char] -> [Int]
ords []      = []
ords (x:xs) = ord x : ords xs
```

Soluție folosind map

```
ords :: [Char] -> [Int]
ords xs = map ord xs
```

Functii de ordin înalt

```
map :: (a -> b) -> [a] -> [b]
map f l = [f x | x <- l]
```

```
Prelude> map ($ 3) [(4 +), (10 *), (^ 2), sqrt]
[7.0, 30.0, 9.0, 1.7320508075688772]
```

În acest caz:

- primul argument este o secțiune a operatorului (\$)
- al doilea argument este o listă de funcții

$$\text{map } (\$ \ x) [f_1, \dots, f_n] == [f_1 \ x, \dots, f_n \ x]$$

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvALb>

Seria 24: <https://questionpro.com/t/AT4NiZvAxq>

Seria 25: <https://questionpro.com/t/AT4qgZvALf>

Selectarea elementelor dintr-o listă - filter

Exemplu - Selectarea elementelor pozitive dintr-o listă

Definiți o funcție care selectează elementele pozitive dintr-o listă.

Soluție descriptivă

```
positives :: [Int] -> [Int]
positives xs = [ x | x <- xs, x > 0 ]
```

Soluție recursivă

```
positives :: [Int] -> [Int]
positives []           = []
positives (x:xs) | x > 0   = x : positives xs
                | otherwise = positives xs
```

```
Prelude> positives [1,-2,3]
[1,3]
```

Exemplu - Selectarea cifrelor dintr-un sir de caractere

Definiți o funcție care selectează cifrele dintr-un sir de caractere.

Soluție descriptivă

```
digits :: [Char] -> [Char]
digits xs = [ x | x <- xs, isDigit x ]
```

Soluție recursivă

```
digits :: [Char] -> [Char]
digits [] = []
digits (x:xs) | isDigit x = x : digits xs
              | otherwise = digits xs
```

```
Prelude> digits "a2c3"
"23"
```

Functia filter

Date fiind un predicat (funcție booleană) și o listă,
selectați elementele din listă care satisfac predicatul.

Soluție descriptivă

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs , p x ]
```

Soluție recursivă

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x = x : filter p xs
                | otherwise = filter p xs
```

Exemplu — Selectarea elementelor pozitive dintr-o listă

Soluție descriptivă

```
positives :: [Int] -> [Int]
positives xs = [ x | x <- xs, x > 0 ]
```

Soluție recursivă

```
positives :: [Int] -> [Int]
positives []           = []
positives (x:xs) | x > 0 = x : positives xs
                | otherwise = positives xs
```

Soluție folosind filter

```
positives :: [Int] -> [Int]
positives xs = filter pos xs
              where pos x = x > 0
```

Exemplu — Selectarea cifrelor dintr-un sir de caractere

Soluție descriptivă

```
digits :: [Char] -> [Char]
digits xs = [ x | x <- xs, isDigit x ]
```

Soluție recursivă

```
digits :: [Char] -> [Char]
digits [] = []
digits (x:xs) | isDigit x = x : digits xs
              | otherwise = digits xs
```

Soluție folosind filter

```
digits :: [Char] -> [Char]
digits xs = filter isDigit xs
```

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvALr>

Seria 24: <https://questionpro.com/t/AT4NiZvAxv>

Seria 25: <https://questionpro.com/t/AT4qgZvALs>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C04

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Currying

Currying

Currying este procedeul prin care o funcție cu mai multe argumente este transformată într-o funcție care are un singur argument și întoarce o altă funcție.

- În Haskell toate funcțiile sunt în forma **curry**, deci au un singur argument.
- Operatorul \rightarrow pe tipuri este asociativ la dreapta, adică tipul $a_1 \rightarrow a_2 \rightarrow \dots \rightarrow a_n$ îl gândim ca $a_1 \rightarrow (a_2 \rightarrow \dots (a_{n-1} \rightarrow a_n) \dots)$.
- Aplicarea funcțiilor este asociativă la stânga, adică expresia $f\ x_1 \dots x_n$ o gândim ca $(\dots ((f\ x_1)\ x_2) \dots x_n)$.

Functiile curry si uncurry si multimi

```
Prelude> :t curry
```

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

```
Prelude> :t uncurry
```

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

Exemplu:

```
f :: (Int, String) -> String
```

```
f (n,s) = take n s
```

```
Prelude> let cf = curry f
```

```
Prelude> :t cf
```

```
cf :: Int -> String -> String
```

```
Prelude> cf (1, "abc")
```

```
"a"
```

```
Prelude> cf 1 "abc"
```

```
"a"
```

Functii în matematică

- Fie $f : A \times B \rightarrow C$ o funcție. În mod ușual scriem $f(x, y) = z$ unde $x \in A$, $y \in B$ și $z \in C$.
- Pentru $x \in A$ (arbitrar, fixat) definim $f_x : B \rightarrow C$, $f_x(y) = z$ dacă și numai dacă $f(x, y) = z$. Funcția f_x se obține prin aplicarea parțială a funcției f .
- Dacă notăm $B \rightarrow C \stackrel{not}{=} \{h : B \rightarrow C \mid h \text{ funcție}\}$ observăm că $f_x \in B \rightarrow C$ pentru orice $x \in A$.
- Asociem lui f funcția $cf : A \rightarrow (B \rightarrow C)$, $cf(x) = f_x$.
- Vom spune că funcția cf este *forma curry* a funcției f .

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvHu0>

Seria 24: <https://questionpro.com/t/AT4NiZu3KB>

Seria 25: <https://questionpro.com/t/AT4qgZvHuP>

Agregarea elementelor dintr-o listă - fold

Exemplu - Suma

Definiți o funcție care dată fiind o listă de numere întregi calculează suma elementelor din listă.

Soluție recursivă

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

```
Prelude> sum [1,2,3,4]
```

10

Exemplu - Produs

Definiți o funcție care dată fiind o listă de numere întregi calculează produsul elementelor din listă.

Soluție recursivă

```
product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs
```

```
Prelude> product [1,2,3,4]
```

24

Exemplu - Concatenare

Definiți o funcție care concatenează o listă de liste.

Soluție recursivă

```
concat :: [[a]] -> [a]
concat []      = []
concat (xs:xss) = xs ++ concat xss
```

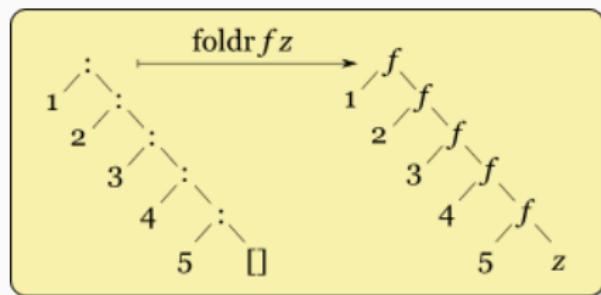
```
Prelude> concat [[1,2,3],[4,5]]
[1,2,3,4,5]
```

```
Prelude> concat ["con","ca","te","na","re"]
"concatenare"
```

Functia foldr

foldr :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare inițială, și o listă, calculați valoarea obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.



Functia foldr

foldr :: (a → b → b) → b → [a] → b

Soluție recursivă

foldr :: (a → b → b) → b → [a] → b

foldr f i [] = i

foldr f i (x:xs) = f x (**foldr** f i xs)

Soluție recursivă cu operator infix

foldr :: (a → b → b) → b → [a] → b

foldr op i [] = i

foldr op i (x:xs) = x `op` (**foldr** f i xs)

Exemplu — Suma

Soluție recursivă

```
sum :: [Int] -> Int
sum []      = 0
sum (x:xs) = x + sum xs
```

Soluție folosind foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

```
sum :: [Int] -> Int
sum xs = foldr (+) 0 xs
```

Exemplu

foldr (+) 0 [1, 2, 3] == 1 + (2 + (3 + 0))

Exemplu — Produs

Soluție recursivă

```
product :: [Int] -> Int
product []      = 1
product (x:xs) = x * product xs
```

Soluție folosind foldr

foldr :: (a -> b -> b) -> b -> [a] -> b

```
product :: [Int] -> Int
product xs = foldr (*) 1 xs
```

Exemplu

foldr (*) 1 [1, 2, 3] == 1 * (2 * (3 * 1))

Exemplu — Concatenare

Soluție recursivă

```
concat :: [[a]] -> [a]
concat []      = []
concat (xs:xss) = xs ++ concat xss
```

Soluție folosind foldr

```
foldr :: (a -> b -> b) -> b -> [a] -> b
concat :: [Int] -> Int
concat xs = foldr (++) [] xs
```

Exemplu

```
foldr (++) [] ["Ana ", "are ", "mere."]
== "Ana " ++ ("are " ++ ("mere." ++ []))
```

Exemplu – Suma pătratelor numerelor pozitive

```
f :: [Int] -> Int
f xs = sum (squares (positives xs))
```

```
f :: [Int] -> Int
f xs = sum [ x*x | x <- xs, x > 0 ]
```

```
f :: [Int] -> Int
f [] = 0
f (x:xs) | x > 0 = (x*x) + f xs
         | otherwise = f xs
```

Exemplu – Suma pătratelor numerelor pozitive

```
f :: [Int] -> Int
```

```
f xs = foldr (+) 0 (map sqr (filter pos xs))
```

where

```
sqr x = x * x
```

```
pos x = x > 0
```

```
f :: [Int] -> Int
```

```
f xs = foldr (+) 0
```

```
(map (\ x -> x * x)
```

```
(filter (\ x -> x > 0) xs))
```

```
f :: [Int] -> Int
```

```
f xs = foldr (+) 0 (map (^2) (filter (>0) xs))
```

```
f :: [Int] -> Int
```

```
f = foldr (+) 0 . map (^2) . filter (>0)
```

Exemplu - Componerea funcțiilor

În definiția lui **foldr**

foldr :: (a → b → b) → b → [a] → b

b poate fi tipul unei funcții.

compose :: [a → a] → (a → a)

compose = **foldr** (.) **id**

Prelude> compose [(+1), (^2)] 3

10

-- *functia (foldr (.) id [(+1), (^2)]) aplicata lui 3*

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvALu>

Seria 24: <https://questionpro.com/t/AT4NiZvIWL>

Seria 25: <https://questionpro.com/t/AT4qgZvALw>

Foldr și Foldl

foldr și foldl

Date fiind o funcție de actualizare a valorii calculate cu un element curent, o valoare initială, și o listă, calculați valoarea obținută prin aplicarea repetată a funcției de actualizare fiecărui element din listă.

foldr :: $(a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

foldr op z [a₁, a₂, a₃, ..., a_n] =
a₁ `op` (a₂ `op` (a₃ `op` (... (a_n `op` z) ...)))

foldl :: $(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

foldl op z [a₁, a₂, a₃, ..., a_n] =
(... (((z `op` a₁) `op` a₂) `op` a₃) ...) `op` a_n

foldr și foldl

Funcția **foldr**

```
foldr :: (a -> b -> b) -> b -> [a] -> b  
foldr f i [] = i  
foldr f i (x:xs) = f x (foldr f i xs)
```

Funcția **foldl**

```
foldl :: (b -> a -> b) -> b -> [a] -> b  
foldl h i [] = i  
foldl h i (x:xs) = foldl h (h i x) xs
```

Suma elementelor dintr-o listă

Definiți o funcție care dată fiind o listă de numere întregi calculează suma elementelor din listă.

Soluție cu foldr

sum = foldr (+) 0

Cu **foldr**, elementele sunt procesate de la dreapta la stânga:

$$\text{sum } [x_1, \dots, x_n] = (x_1 + (x_2 + \dots (x_n + 0) \dots))$$

Suma elementelor dintr-o listă

Solutie in care elementele sunt procesate de la stânga la dreapta.

```
sum :: [Int] -> Int
```

```
sum xs = suml xs 0
```

where

```
suml [] n = n
```

```
suml (x:xs) n = suml xs (n+x)
```

Elementele sunt procesate de la stânga la dreapta:

$$\text{suml } [x_1, \dots, x_n] 0 = (\dots (0 + x_1) + x_2) + \dots x_n$$

Soluție cu foldl

```
sum :: [Int] -> Int
```

```
sum xs = foldl (+) 0 xs
```

Inversarea elementelor unei liste

Definiți o funcție care dată fiind o listă de elemente, calculează lista în care elementele sunt scrise în ordine inversă.

Soluție cu foldl

```
--      flip  :: (a -> b -> c) -> (b -> a -> c)
--      (:)  :: a -> [a] -> [a]
-- flip (:) :: [a] -> a -> [a]

rev = foldl (<:>) []
            where (<:>) = flip (:)
```

Elementele sunt procesate de la stânga la dreapta:

$$\text{rev } [x_1, \dots, x_n] = (\dots(([] <:> x_1) <:> x_2) \dots) <:> x_n$$

Evaluare lenesă. Liste infinite

Putem folosi funcțiile **map** și **filter** pe liste infinite:

```
Prelude> inf = map (+10) [1..] -- inf nu este evaluat
Prelude> take 3 inf
[11,12,13]
```

Limbajul Haskell folosește implicit evaluarea lenesă

- expresiile sunt evaluate numai când este nevoie de valoarea lor
- expresiile nu sunt evaluate total, elementele care nu sunt folosite rămân neevaluate
- o expresie este evaluată o singură dată.

În exemplul de mai sus, este acceptată definiția lui `inf`, fără a fi evaluată. Când expresia `take 3 inf` este evaluată, numai primele 3 elemente ale lui `inf` sunt calculate, restul rămânând neevaluate.

Evaluare lenășă: lista numerelor prime

Vă amintiți din primul curs:

```
primes = sieve [2..]
```

```
sieve (p:ps) = p : sieve [ x | x <- ps, mod x p /= 0 ]
```

Intuitiv, evaluarea lenășă funcționează astfel:

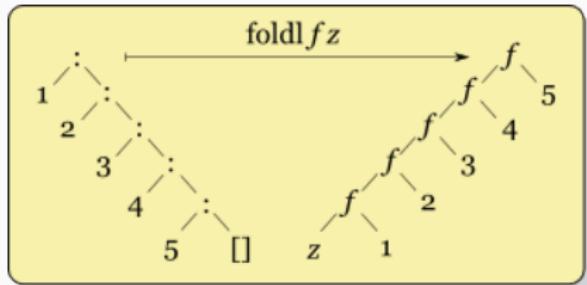
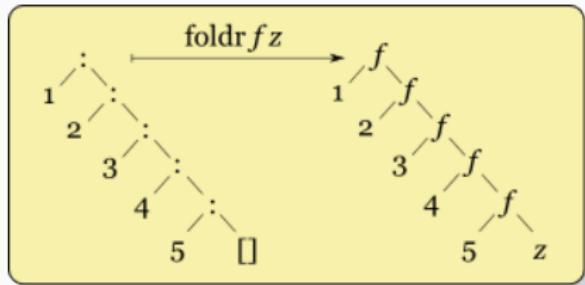
```
sieve [2..] -->
```

```
2 : sieve [ x | x <- [3..], mod x 2 /= 0 ] -->
```

```
2 : sieve (3 : [ x | x <- [4..], mod x 2 /= 0 ]) -->
```

```
2 : 3 : sieve ([ y | y <-
                  [x | x <- [4..], mod x 2 /= 0 ],
                  mod y 3 /= 0 ])
--> ...
```

foldr și foldl



[https://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](https://en.wikipedia.org/wiki/Fold_(higher-order_function))

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri)
- **foldl** nu poate fi folosită pe liste infinite niciodată

foldr și foldl

- **foldr** poate fi folosită pe liste infinite (în anumite cazuri),
- **foldl** nu poate fi folosită pe liste infinite niciodată.

```
Prelude> foldr (*) 0 [1..]
```

*** Exception: stack overflow

```
Prelude> take 3 $ foldr (\x xs-> (x+1):xs) [] [1..]
```

[2,3,4]

-- foldr a functionat pe o lista infinită

```
Prelude> take 3 $ foldl (\xs x-> (x+1):xs) [] [1..]
```

-- expresia se calculeaza la infinit

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvHue>

Seria 24: <https://questionpro.com/t/AT4NiZvIW4>

Seria 25: <https://questionpro.com/t/AT4qgZvHuf>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C05

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**

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

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

Tipuri produs

Problema. 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
```

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)
```

- **List** este constructor de tip
- Nil și 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.

Forma generală:

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

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

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

Tipuri de date algebrice

Forma generală:

```
data Typename = Cons1 t11 ... t1k1
          | Cons2 t21 ... t2k2
          | ...
          | Consn tn1 ... tnkn
```

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

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
```

```
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)
```

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvPr7>

Seria 24: <https://questionpro.com/t/AT4NiZvQqv>

Seria 25: <https://questionpro.com/t/AT4qgZvPr8>

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)

Declarație ca tip de date algebraic folosind şabloane

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

Adunarea pe tipul de date algebraic:

```
(++) :: Nat -> Nat -> Nat
```

```
m +++ Zero      = m
```

```
m +++ (Succ n) = Succ (m +++ n)
```

Comparați cu versiunea folosind notația predefinită:

```
(+) :: Int -> Int -> Int
```

```
m + 0 = m
```

```
m + n = (m + (n-1)) + 1
```

Exemplu: numerele naturale (Peano)

Declarație ca tip de date algebric folosind şabloane

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

```
(***) :: Nat -> Nat -> Nat
```

```
m *** Zero = Zero
```

```
m *** (Succ n) = (m *** n) +++ m
```

Comparați cu versiunea folosind notația predefinită:

```
(*) :: Int -> Int -> Int
```

```
m * 0 = 0
```

```
m * n = (m * (n-1)) + m
```

Tipul Maybe (opțiune)

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

Rezultate optionale

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

Argumente optionale

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

Maybe - folosirea unui rezultat optional

```
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
```

-- utilizare *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
```

```
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]
```

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 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.

```
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. Proiectii

```
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 }
```

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**.

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C06

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Anunt - examen parțial

- valorează **3 puncte** din nota finală
- durata **40 min**
- în săptămâna 7, **in cadrul cursului**
- **test grila pe foaie**
- nu este obligatoriu și nu se poate reface
- va conține **30 de întrebări grilă** asemănatoare cu cele din curs
- veți găsi repartitia pentru parțial pe Drive
(unde aveți suporturile de curs/laborator)
- materiale ajutătoare: suporturile de curs și de laborator, format fizic sau pe laptop/tableta
- notitele voastre sunt permise
- nu aveți voie cu materialele pe telefoane mobile, ceasuri
- va ruga să va asigurați că aveți laptopurile încărcate/că aveți prelungitoare pentru a va încărca laptopurile

Clase de tipuri

Exemplu: test de apartenență

Să scriem funcția **my_elem** care testează dacă un element aparține unei liste.

- definiția folosind descrieri de liste

```
my_elem x ys      = or [ x == y | y <- ys ]
```

- definiția folosind recursivitate

```
my_elem x []      = False
```

```
my_elem x (y:ys) = x == y || my_elem x ys
```

- definiția folosind funcții de nivel înalt

```
my_elem x ys      = foldr (||) False (map (x ==) ys)
```

Funcția my_elem este polimorfică

Prelude> my_elem 1 [2,3,4]

False

Prelude> my_elem 'o' "word"

True

Prelude> my_elem (1,'o') [(0,'w'),(1,'o'),(2,'r')]

True

Prelude> my_elem "word" ["list","of","word"]

True

Care este tipul funcției my_elem?

Funcția my_elem este **polimorfică**.

Definiția funcției este parametrică în tipul de date.

Functia my_elem este polimorfică

Totuși definiția nu funcționează pentru orice tip!

```
Prelude> my_elem (+ 2) [(+ 2), sqrt]
```

```
No instance for (Eq (Double -> Double)) arising from a use of 'my_elem'
```

Ce se întâmplă?

```
Prelude> :t my_elem
```

```
my_elem :: Eq a => a -> [a] -> Bool
```

În definiția

```
my_elem x ys = or [ x == y | y <- ys ]
```

folosim relația de egalitate == care nu este definită pentru orice tip.

```
Prelude> sqrt == sqrt
```

```
No instance for (Eq (Double -> Double)) ...
```

```
Prelude> ("ab",1) == ("ab",2)
```

```
False
```

Clase de tipuri

O clasă de tipuri este determinată de o mulțime de funcții (este o interfață).

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
  -- ^^^ putem avea definiții implicite
```

Puteti verifica folosind comanda `:info` sau `:i` ce conține o anumită clasă de tipuri.

Clase de tipuri

Tipurile care aparțin clasei sunt instanțe ale clasei.

```
instance Eq Bool where
    False == False = True
    False == True  = False
    True  == False = False
    True  == True  = True
```

Clasa de tipuri. Constrângeri de tip

În semnatura funcției my_elem trebuie să precizăm ca tipul a este în clasa **Eq**

```
my_elem :: Eq a => a -> [a] -> Bool
```

- **Eq** a se numește **constrângere de tip**.
- **=>** separă constrângerile de tip de restul signaturii.

În exemplul de mai sus am considerat că my_elem este definită pe liste, dar în realitate funcția este mai complexă:

```
Prelude> :t my_elem
my_elem :: (Eq a, Foldable t) => a -> t a -> Bool
```

În această definiție Foldable este o altă clasă de tipuri, iar t este un parametru care ține locul unui **constructor de tip**!

Instanțe ale lui Eq

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq Int      where
  (==) = eqInt    -- built-in

instance Eq Char     where
  x == y = ord x == ord y

instance (Eq a, Eq b) => Eq (a,b) where
  (u,v) == (x,y) = (u == x) && (v == y)

instance Eq a => Eq [a] where
  [] == []      = True
  [] == y:ys   = False
  x:xs == []   = False
  x:xs == y:ys = (x == y) && (xs == ys)
```

Eq, Ord

Clasele pot fi extinse:

```
class (Eq a) => Ord a where
    (<) :: a -> a -> Bool
    (≤) :: a -> a -> Bool
    (>) :: a -> a -> Bool
    (≥) :: a -> a -> Bool
    -- minimum definition : (≤)
    x < y = x ≤ y && x /= y
    x > y = y < x
    x ≥ y = y ≤ x
```

Clasa **Ord** este clasa tipurilor de date înzestrate cu o relație de ordine.

În definiția clasei **Ord** s-a impus o constrângere de tip. Astfel, orice instanță a clasei **Ord** trebuie să fie instanță a clasei **Eq**.

Instanțe ale lui Ord

```
instance Ord Bool where
    False <= False = True
    False <= True = True
    True <= False = False
    True <= True = True
```

```
instance (Ord a, Ord b) => Ord (a,b) where
    (x,y) <= (x',y') = x < x' || (x == x' && y <= y')
    -- ordinea lexicografica
```

```
instance Ord a => Ord [a] where
    [] <= ys = True
    (x:xs) <= [] = False
    (x:xs) <= (y:ys) = x < y || (x == y && xs <= ys)
```

Definirea claselor

Să presupunem că vrem să definim o clasă de tipuri pentru datele care pot fi afișate. O astfel de clasă trebuie să conțină o metodă care să indice modul de afișare:

```
class Visible a where
    toString :: a -> String
```

Putem face instantieri astfel:

```
instance Visible Char where
    toString c = [c]
```

Clasele **Eq**, **Ord** sunt predefinite. Clasa **Visible** este definită de noi, dar există o clasă predefinită care are același rol: clasa **Show**.

Show

```
class Show a where
  show :: a -> String      -- analogul lui "toString"

instance Show Bool where
  show False      = "False"
  show True       = "True"

instance (Show a, Show b) => Show (a,b) where
  show (x,y) = "(" ++ show x ++ "," ++ show y ++ ")"

instance Show a => Show [a] where
  show []        = "[]"
  show (x:xs)   = "[" ++ showSep x xs ++ "]"
    where
      showSep x []     = show x
      showSep x (y:ys) = show x ++ ", " ++ showSep y ys
```

Constructori simboluri

```
data List a = Nil
           | a :: List a
infixr 5 :::

eqList :: Eq a => List a -> List a -> Bool
eqList Nil Nil          = True
eqList (x :: xs) (y :: ys) = x == y && eqList xs ys
eqList _ _                = False

instance (Eq a) => Eq (List a) where
  (==) = eqList
```

Constructori simboluri

```
data List a = Nil
           | a :: List a
infixr 5 :::

showMyList :: Show a => List a -> String
showMyList Nil = "Nil"
showMyList (x :: xs) = show x ++ " :: " ++
                      showMyList xs

instance (Show a) => Show (List a) where
    show = showMyList
```

Clase de tipuri pentru numere

```
class (Eq a, Show a) => Num a where
    (+), (-), (*) :: a -> a -> a
    negate :: a -> a
    ...
    fromInteger :: Integer -> a
    -- minimum definition: (+), (-), (*), fromInteger
    negate x       = fromInteger 0 - x
```

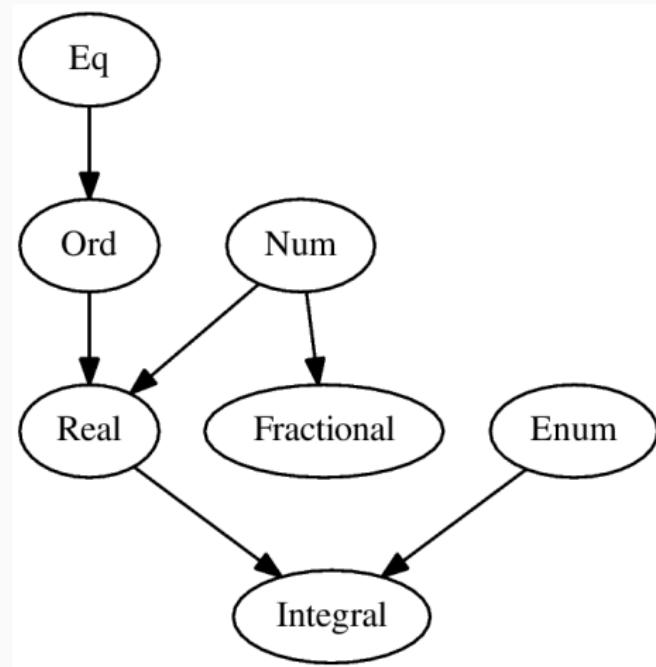
```
class (Num a) => Fractional a where
    (/)           :: a -> a -> a
    recip         :: a -> a
    fromRational :: Rational -> a
    ...
    -- minimum definition: (/), fromRational
    recip x      = 1/x
```

Clase de tipuri pentru numere

```
class (Num a, Ord a) => Real a where
    toRational      :: a -> Rational
    ...
    ...

class (Real a, Enum a) => Integral a where
    div, mod        :: a -> a -> a
    toInteger       :: a -> Integer
    ...
    ...
```

Clase de tipuri



sursa: C. Allen, J. Moronuki - "Haskell Programming from first principles"

Derivare automată pentru tipuri algebrice

Am definit tipuri de date noi:

```
data Season = Spring | Summer | Autumn | Winter  
           deriving (Eq, Ord, Show)
```

```
data Point a b = Pt a b  
               deriving (Eq, Ord, Show)
```

Cum putem să le facem instanțe ale claselor **Eq, Ord, Show?**

Putem să le facem explicit sau să folosim derivarea automată.

Atenție! Derivarea automată poate fi folosită numai pentru unele clase predefinite.

Derivare automată vs Instanțiere explicită

Instanțierea prin derivare automată:

```
data Point a b = Pt a b  
                deriving Eq
```

Instanțiere explicită:

```
instance Eq a => Eq (Point a b) where  
  (==) (Pt x1 y1) (Pt x2 y2) = (x1 == x2)
```

Derivare automata pentru tipuri algebrice

```
data Point a b = Pt a b  
    deriving (Eq, Ord, Show)
```

Egalitatea, relația de ordine și modalitatea de afișare sunt definite implicit dacă este posibil:

```
Prelude> Pt 2 3 < Pt 5 6  
True
```

```
Prelude> Pt 2 "b" < Pt 2 "a"  
False
```

```
Prelude> Pt (+2) 3 < Pt (+5) 6
```

No instance for (Ord (Integer -> Integer)) arising from a use of '<'

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvXgU>

Seria 24: <https://questionpro.com/t/AT4NiZvW58>

Seria 25: <https://questionpro.com/t/AT4qgZvXgW>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C07-C08

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Functori

Map

Va aduceti aminte functia

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

Problema. Putem generaliza aceasta functie la alte tipuri parametrizate?

Clasa de tipuri Functor

Definiție

```
class Functor f where  
    fmap :: (a -> b) -> f a -> f b
```

Dată fiind o funcție $f :: a \rightarrow b$ și ca $:: f a$, $fmap$ produce $cb :: f b$ obținută prin transformarea rezultatelor produse de computația ca folosind funcția f (și doar atât!)

Instanță pentru liste

```
instance Functor [] where  
    fmap = map
```

Clasa de tipuri Functor

```
class Functor f where  
  fmap :: (a -> b) -> f a -> f b
```

Instanță pentru tipul optiune

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

```
instance Functor Maybe where  
  fmap f Nothing = Nothing  
  fmap f (Just x) = Just (f x)
```

Clasa de tipuri Functor

```
class Functor f where  
    fmap :: (a -> b) -> f a -> f b
```

Instanță pentru tipul eroare

```
fmap :: (a -> b) -> Either e a -> Either e b
```

```
instance Functor (Either e) where  
    fmap _ (Left x) = Left x  
    fmap f (Right y) = Right (f y)
```

Clasa de tipuri Functor

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

```
data Arboare a = Nil
                | Nod a Arboare Arboare
```

Instantă pentru tipul arbore

fmap :: (a -> b) -> Arboare a -> Arboare b

instance Functor Arboare where

fmap f Nil = Nil

fmap f (Nod x l r) = Nod (f x) (fmap f l) (fmap f r)

Clasa de tipuri Functor

```
class Functor f where  
    fmap :: (a -> b) -> f a -> f b
```

Tipul funcțiilor de sursă dată $t \rightarrow a$ (parametric in a)

Instanță pentru tipul funcție

$fmap :: (a \rightarrow b) \rightarrow (t \rightarrow a) \rightarrow (t \rightarrow b)$

instance Functor (->) a where

$fmap f g = f . g$ -- sau, mai simplu, $fmap = (.)$

Exemple

```
Prelude> fmap (*2) [1..3]
[2,4,6]
Prelude> fmap (*2) (Just 200)
Just 400
Prelude> fmap (*2) Nothing
Nothing
Prelude> fmap (*2) (+100) 4
208
Prelude> fmap (*2) (Right 6)
Right 12
Prelude> fmap (*2) (Left 135)
Left 135
Prelude> (fmap . fmap) (+1) [Just 1, Just 2, Just 3]
[Just 2, Just 3, Just 4]
```

Proprietăți ale functorilor

- Argumentul **f** al lui **Functor f** definește o transformare de tipuri
 - **f** a este tipul a transformat prin functorul **f**
- **fmap** definește transformarea corespunzătoare a funcțiilor
 - **fmap :: (a -> b) -> (f a -> f b)**

Contractul lui **fmap**

- **fmap f ca** e obținută prin transformarea rezultatelor produse de computația ca folosind funcția **f** (și doar atât!)
- Abstractizat prin două legi:
 - identitate** $\text{fmap id} == \text{id}$
 - componere** $\text{fmap (g . h)} == \text{fmap g . fmap h}$

Invalidarea contractului - identitate

```
data WhoCares a = ItDoesnt
  | Matter a
  | WhatThisIsCalled
deriving (Eq, Show)
```

Instanta a clasei Functor care invalideaza conditia de conservare a identitatii:

```
instance Functor WhoCares where
  fmap _ ItDoesnt = WhatThisIsCalled
  fmap _ WhatThisIsCalled = ItDoesnt
  fmap f (Matter a) = Matter (f a)
```

```
Prelude> fmap id ItDoesnt
WhatThisIsCalled
Prelude> id ItDoesnt
ItDoesnt
```

Validarea contractului - identitate

```
data WhoCares a = ItDoesnt
    | Matter a
    | WhatThisIsCalled
deriving (Eq, Show)
```

Instanta a clasei Functor care valideaza conditia de conservare a identitatii:

```
instance Functor WhoCares where
    fmap _ ItDoesnt = ItDoesnt
    fmap _ WhatThisIsCalled = WhatThisIsCalled
    fmap f (Matter a) = Matter (f a)
```

```
Prelude> fmap id ItDoesnt
```

```
ItDoesnt
```

```
Prelude> id ItDoesnt
```

```
ItDoesnt
```

Invalidarea contractului - compunere

```
data CountingBad a =  
    Heisenberg Int a  
deriving (Eq, Show)
```

Instanta a clasei Functor care invalideaza conditia de conservare a compunerii:

```
instance Functor CountingBad where  
    fmap f (Heisenberg n a) = Heisenberg (n+1) (f a)
```

```
Prelude> oneWhoKnocks = Heisenberg 0 "Uncle"
```

```
Prelude> f = (++ " Jesse")
```

```
Prelude> g = (++ " lol")
```

```
Prelude> fmap (f . g) oneWhoKnocks
```

```
Heisenberg 1 "Uncle lol Jesse"
```

```
Prelude> fmap f . fmap g $ oneWhoKnocks
```

```
Heisenberg 2 "Uncle lol Jesse"
```

Validarea contractului - compunere

```
data CountingBad a =  
    Heisenberg Int a  
deriving (Eq, Show)
```

Instanta a clasei Functor care valideaza conditia de conservare a compunerii:

```
instance Functor CountingBad where  
    fmap f (Heisenberg n a) = Heisenberg n (f a)
```

```
Prelude> oneWhoKnocks = Heisenberg 0 "Uncle"
```

```
Prelude> f = (++ " Jesse")
```

```
Prelude> g = (++ " lol")
```

```
Prelude> fmap (f . g) oneWhoKnocks
```

```
Heisenberg 0 "Uncle lol Jesse"
```

```
Prelude> fmap f . fmap g $ oneWhoKnocks
```

```
Heisenberg 0 "Uncle lol Jesse"
```

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvoR5>

Seria 24: <https://questionpro.com/t/AT4NiZvmKW>

Seria 25: <https://questionpro.com/t/AT4qgZvoR8>

Functori applicativi

Functori - recap

Va aduceti aminte functia

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

Problema. Putem generaliza aceasta functie la alte tipuri parametrizate?

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Problemă

- Folosind **fmap** putem transforma o funcție $h :: a \rightarrow b$ într-o funcție $fmap h :: m a \rightarrow m b$
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente?
E.g., cum trecem de la $h :: a \rightarrow b \rightarrow c$ la
 $h' :: m a \rightarrow m b \rightarrow m c$
- Putem încerca să folosim fmap

Problemă

- Folosind **fmap** putem transforma o funcție $h :: a \rightarrow b$ într-o funcție **fmap h :: m a → m b**
- Dar ce se întâmplă dacă avem o funcție cu mai multe argumente?
E.g., cum trecem de la $h :: a \rightarrow b \rightarrow c$ la
 $h' :: m a \rightarrow m b \rightarrow m c$
- Putem încerca să folosim **fmap**
- Dar, deoarece $h :: a \rightarrow (b \rightarrow c)$, obținem
fmap h :: m a → m (b → c)
- Putem aplica **fmap h** la o valoare ca $:: m a$ și obținem
fmap h ca :: m (b → c)

Problemă

Cum transformăm un obiect din $m(b \rightarrow c)$ într-o funcție
 $m b \rightarrow m c$?

- **ap** :: $m(b \rightarrow c) \rightarrow (m b \rightarrow m c)$, sau, ca operator
- **(<*>)** :: $m(b \rightarrow c) \rightarrow m b \rightarrow m c$

Merge pentru funcții cu oricâte argumente

Problemă

Dată fiind o funcție

$f :: a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a$

și computațiile

$ca_1 :: m\ a_1, ca_2 :: m\ a_2, \dots, ca_n :: m\ a_n,$

vrem să „aplicăm” funcția f pe rând computațiilor ca_1, \dots, ca_n pentru a obține o computație finală $ca :: m\ a$.

Merge pentru funcții cu oricâte argumente

Date fiind

- $f :: a_1 \rightarrow a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a$
- $ca_1 :: m\ a_1, ca_2 :: m\ a_2, \dots, can :: m\ a_n,$
- $fmap :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$
- $(\langle * \rangle) :: m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$ cu „proprietăți bune”

Atunci

$fmap\ f :: m\ a_1 \rightarrow m\ (a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

$fmap\ f\ ca_1 :: m\ (a_2 \rightarrow a_3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

$fmap\ f\ ca_1\ \langle *\rangle\ ca_2 :: m\ (a_3 \rightarrow \dots \rightarrow a_n \rightarrow a)$

...

$fmap\ f\ ca_1\ \langle *\rangle\ ca_2\ \langle *\rangle\ ca_3\ \dots\ \langle *\rangle\ can :: m\ a$

Clasa de tipuri Applicative

```
class Functor m => Applicative m where
    pure :: a -> m a
    (<*>) :: m (a -> b) -> m a -> m b
```

- Orice instanță a lui Applicative trebuie să fie instanță a lui **Functor**
- `pure` transformă o valoare într-o computație minimală care are acea valoare ca rezultat, și nimic mai mult!
- `(<*>)` ia o computație care produce funcții și o computație care produce argumente pentru funcții și obține o computație care produce rezultatele aplicării funcțiilor asupra argumentelor

Clasa de tipuri Applicative

```
class Functor m where
    fmap :: (a -> b) -> m a -> m b

class Functor m => Applicative m where
    pure :: a -> m a
    (<*>) :: m (a -> b) -> m a -> m b
```

Proprietate importantă

- $fmap f x == pure f <*> x$
- Se definește operatorul ($<\$>$) prin $(<\$>) = fmap$

Functori aplicativi

```
($) :: (a -> b) -> a -> b
(<$>) :: (a -> b) -> m a -> m b
(<*>) :: m (a -> b) -> m a -> m b
```

Instante – Maybe

```
class Functor m where
    fmap :: (a -> b) -> m a -> m b

class Functor m => Applicative m where
    pure :: a -> m a
    (<*>) :: m (a -> b) -> m a -> m b

instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    Just f <*> x = fmap f x
```

Instante – Maybe

```
Prelude> pure "Hey" :: Maybe String  
Just "Hey"
```

Cum concatenam **Just** "Hey" cu **Just** "You!"?

Instante – Maybe

```
Prelude> pure "Hey" :: Maybe String  
Just "Hey"
```

Cum concatenam **Just** "Hey" cu **Just** "You!"?

```
Prelude> (++) <$> (Just "Hey ") <*> (Just "You!")  
Just "Hey You!"
```

Instante – Maybe

```
Prelude> pure "Hey" :: Maybe String  
Just "Hey"
```

Cum concatenam **Just** "Hey" cu **Just** "You!"?

```
Prelude> (++) <$> (Just "Hey ") <*> (Just "You!")  
Just "Hey You!"
```

- **(++) :: String -> (String -> String)**
- **Just "Hey"** :: **Maybe String**
- **(<\$>) :: (a -> b) -> m a -> m b** (este fmap)
- **(++) <\$> (Just "Hey") :: Maybe (String -> String)**
- **Just "You!" :: Maybe String**
- **(<*>) :: m (b -> c) -> m b -> m c**
- **Just "Hey>You!" :: Maybe String**

Instante – Maybe

```
mDiv x y = if y == 0 then Nothing  
            else Just (x `div` y)
```

```
mF x = (+) <$> pure 4 <*> mDiv 10 x
```

- $(+)$:: **Int** \rightarrow **Int** \rightarrow **Int**
- $\text{pure } 4$:: **Maybe Int**
- $(<\$>)$:: $(a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(+)<\$> \text{pure } 4$:: **Maybe (Int} \rightarrow **Int**)**
- mDiv :: **Int** \rightarrow **Int** \rightarrow **Maybe Int**
- $\text{mDiv } 10\ x$:: **Maybe Int**
- $(<*>)$:: $m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$

Prelude> mF 2

Just 9

Instante – Either

```
class Functor m where
    fmap :: (a -> b) -> m a -> m b

class Functor m => Applicative m where
    pure :: a -> m a
    (<*>) :: m (a -> b) -> m a -> m b

instance Applicative (Either a) where
    pure = Right
    Left e <*> _ = Left e
    Right f <*> x = fmap f x
```

Instante – Either

```
Prelude> pure "Hey" :: Either a String  
Right "Hey"
```

```
Prelude> (++) <$> (Right "Hey ") <*> (Right "You!")  
Right "Hey You!"
```

- `(++) :: String -> (String -> String)`
- `Right "Hey"` :: `Either a String`
- `(<$>)` :: `(a -> b) -> m a -> m b` (este fmap)
- `(++) <$> (Right "Hey") :: Either a (String -> String)`
- `Right "You!" :: Either a String`
- `(<*>)` :: `m (b -> c) -> m b -> m c`
- `Right "HeyYou!" :: Either a String`

Instante – Either

```
eDiv x y = if y == 0 then Left "Division by 0!"  
            else Right (x `div` y)
```

```
eF x = (+) <$> pure 4 <*> eDiv 10 x
```

- $(+)$:: **Int** \rightarrow **Int** \rightarrow **Int**
- $\text{pure } 4$:: **Either String Int**
- $(<\$>)$:: $(a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(+)<\$> \text{pure } 4$:: **Either String (Int} \rightarrow **Int**)**
- eDiv :: **Int** \rightarrow **Int** \rightarrow **Either String Int**
- $\text{eDiv}\ 10\ x$:: **Either String Int**
- $(<*>)$:: $m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$

```
Prelude> eF 2
```

```
Right 9
```

Instante – Liste

```
class Functor m where
    fmap :: (a -> b) -> m a -> m b

class Functor m => Applicative m where
    pure :: a -> m a
    (<*>) :: m (a -> b) -> m a -> m b

instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Instante – Liste

```
Prelude> pure "Hey" :: [String]
["Hey"]
```

```
Prelude> (++) <$> ["Hello ", "Goodbye "] <*> ["world"
, "happiness"]
["Hello world", "Hello happiness", "Goodbye world", "
Goodbye happiness"]
```

- $(++) :: \text{String} \rightarrow (\text{String} \rightarrow \text{String})$
- $["Hello", "Goodbye"] :: [\text{String}]$
- $(<\$>) :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(++) <\$> ["Hello", "Goodbye"] :: [\text{String} \rightarrow \text{String}]$
- $["world", "happiness"] :: [\text{String}]$
- $(<*>) :: m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$

Instante – Liste

```
Prelude> (+) <$> [1,2] <*> [3,4]  
[4,5,5,6]
```

- `(+)` :: `Int` → `Int` → `Int`
- `[1,2]` :: `[Int]`
- `(<$>)` :: $(a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este `fmap`)
- `(<*>)` :: $m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$
- `[(+),(*)] <*> [1,2]` :: `[Int → Int]`
- `[(+),(*)] <*> [1,2] <*> [3,4]` :: `[Int]`

Instante – Liste

```
Prelude> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

- $(+),(*) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
- $[(+),(*)] :: [\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}]$
- $[1,2] :: [\text{Int}]$
- $(<*>) :: m(b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$
- $[(+),(*)] <*> [1,2] :: [\text{Int} \rightarrow \text{Int}]$
- $[(+),(*)] <*> [1,2] <*> [3,4] :: [\text{Int}]$

Instante – Liste

```
Prelude> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]  
[55,80,100,110]
```

- $(*) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$
- $[2,5,10] :: [\text{Int}]$
- $(<\$>) :: (a \rightarrow b) \rightarrow m\ a \rightarrow m\ b$ (este fmap)
- $(*) <\$> [2,5,10] :: [\text{Int} \rightarrow \text{Int}]$
- $(<*>) :: m\ (b \rightarrow c) \rightarrow m\ b \rightarrow m\ c$
- $(*) <\$> [2,5,10] <*> [8,10,11] :: [\text{Int}]$

```
Prelude> (*) <$> [2,5,10] <*> [8,10,11]  
[16,20,22,40,50,55,80,100,110]
```

Proprietăți ale functorilor aplicativi

```
class Functor m where
```

```
fmap :: (a -> b) -> m a -> m b
```

```
class Functor m => Applicative m where
```

```
pure :: a -> m a
```

```
(<*>) :: m (a -> b) -> m a -> m b
```

- identitate pure **id** $\text{pure id } v = v$
- compozitie pure $(.)$ $\text{pure } u \text{ } <*> \text{pure } v = \text{pure } (u \text{ } <*> v)$
- homomorfism $\text{pure } f \text{ } <*> \text{pure } x = \text{pure } (f \text{ } x)$

Consecință: $\text{fmap } f \text{ } x == f \text{ } <\$> \text{ } x == \text{pure } f \text{ } <*> \text{ } x$

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvoSC>

Seria 24: <https://questionpro.com/t/AT4NiZvmNw>

Seria 25: <https://questionpro.com/t/AT4qgZvoSD>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C09

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monoid

din nou foldr

```
foldr :: (a -> b -> b) -> b -> t a -> b
```

```
Prelude> foldr (+) 0 [1,2,3]
```

```
6
```

```
Prelude> foldr (*) 1 [1,2,3]
```

```
6
```

```
Prelude> foldr (++) [] ["1","2","3"]
```

```
"123"
```

```
Prelude> foldr (||) False [True, False, True]
```

```
True
```

```
Prelude> foldr (&&) True [True, False, True]
```

```
False
```

Ce au in comun aceste operații?

Monoizi

(M, \circ, e) este **monoid** dacă

- $\circ : M \times M \rightarrow M$ este asociativă
- $m \circ e = e \circ m = m$, oricare $m \in M$

Exemple de monoizi:

$(\text{Int}, +, 0)$, $(\text{Int}, *, 1)$, $(\text{String}, ++, [])$,
 $(\{\text{True}, \text{False}\}, \&\&, \text{True})$, $(\{\text{True}, \text{False}\}, \|\|, \text{False})$

Operația de monoid poate fi generalizată pe liste:

```
sum = foldr (+) 0
product = foldr (*) 1
concat = foldr (++) []
and = foldr (\&&) True
or = foldr (\|\|) False
```

Monoizi și semigrupuri

(M, \circ, e) este monoid dacă

- $\circ : M \times M \rightarrow M$ este asociativă
- $m \circ e = e \circ m = m$, oricare $m \in M$

Un semigrup este un monoid fără element neutru.

(M, \circ) este semigrup dacă

- $\circ : M \times M \rightarrow M$ este asociativă.

Exemple

- Orice monoid este și semigrup
- Semigrupul numerelor naturale pozitive, cu adunarea $(\mathbb{N}^*, +)$
- Semigrupul numerelor intregi nenule, cu înmulțirea $(\mathbb{Z}^*, *)$
- Semigrupul listelor nevide, cu concatenarea

clasele Semigroup și Monoid

```
class Semigroup a where
    (<>) :: a -> a -> a      -- operatia asociativa
infixr 6 <>

class Semigroup a => Monoid a where
    mempty :: a                  -- elementul neutru

    mconcat :: [a] -> a         -- generalizarea la liste
    mconcat = foldr (<>) mempty
```

Legi

- Asociativitate: $x \text{ <>} (y \text{ <>} z) = (x \text{ <>} y) \text{ <>} z$
- Identitate la dreapta: $x \text{ <>} \text{mempty} = x$
- Identitate la stânga: $\text{mempty} \text{ <>} x = x$
- Atenție! Acest lucru este responsabilitatea programatorului!

clasa Monoid

Instanta pentru liste

```
instance Semigroup [a] where
    (⟨⟩) = (++)
instance Monoid [a] where
    mempty = []
```

```
Prelude> mempty :: [a]
[]
Prelude> mconcat [[1 ,2 ,3] ,[4 ,5] ,[6]]
[1 ,2 ,3 ,4 ,5 ,6]
```

newtype

(Int, +, 0), (Int, *, 1) sunt monoizi

({True,False}, &&, True), ({True,False}, ||, False) sunt monoizi

Cum definim instante diferite pentru acelasi tip?

- se crează o copie a tipului folosind **newtype**
- copia este definită ca instantă a tipului

newtype Nat = MkNat Integer

- **newtype** se folosește cand un singur constructor este aplicat unui singur tip de date
- declarația cu **newtype** este mai eficientă decât cea cu **data**
- **type** redenumește tipul; **newtype** face o copie și permite redefinirea operațiilor

clasa Monoid

Bool ca monoid față de conjuncție

```
newtype All = All { getAll :: Bool }
    deriving (Eq, Show)
instance Semigroup All where
    All x <>> All y = All (x && y)
instance Monoid All where
    mempty = All True
```

Bool ca monoid față de disjuncție

```
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Show)
instance Semigroup Any where
    Any x <>> Any y = Any (x || y)
instance Monoid Any where
    mempty = Any False
```

clasa Monoid

Num a ca monoid față de adunare

```
newtype Sum a = Sum { getSum :: a }
    deriving (Eq, Show)
instance Num a => Semigroup (Sum a) where
    Sum x <>> Sum y = Sum (x + y)
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
```

Num a ca monoid față de înmulțire

```
newtype Product a = Product { getProduct :: a }
    deriving (Eq, Show)
instance Num a => Semigroup (Product a) where
    Product x <>> Product y = Product (x * y)
instance Num a => Monoid (Product a) where
    mempty = Product 1
```

clasa Monoid

Ord a ca semigrup față de operația de minim

```
newtype Min a = Min { getMin :: a }
    deriving (Eq, Show)
instance Ord a => Semigroup (Min a) where
    Min x <>> Min y = Min (min x y)
instance (Ord a, Bounded a) => Monoid (Min a) where
    mempty = Min maxBound
```

Ord a ca semigrup față de operația de maxim

```
newtype Max a = Max { getMax :: a }
    deriving (Eq, Show)
instance Ord a => Semigroup (Max a) where
    Max x <>> Max y = Max (max x y)
instance (Ord a, Bounded a) => Monoid (Max a) where
    mempty = Max minBound
```

clasa Monoid

```
Prelude> Sum 3
Sum {getSum = 3}
Prelude> Sum 3 <> Sum 4
Sum {getSum = 7}
Prelude> Product 3 <> Product 4
Product {getProduct = 12}
Prelude> mconcat [Any False,Any True,Any False]
Any {getAny = True}
Prelude> (getSum . mconcat) [Sum 3,Sum 4,Sum 5]
12
Prelude> getMax . mconcat . map Max $ [3,5,4]
5
```

Monoid Maybe

```
instance Semigroup a => Semigroup (Maybe a) where
    Nothing <>> m          =  m
    m        <>> Nothing     =  m
    Just m1 <>> Just m2     =  Just (m1 <>> m2)
```

```
instance Semigroup a => Monoid (Maybe a) where
    mempty = Nothing
```

```
Prelude> Nothing <>> (Just 3) :: Maybe Integer
<interactive>:35:1: error:
```

```
Prelude> Nothing <>> (Just (Sum 3))
Just (Sum {getSum = 3})
```

Semigroup

Tipul listelor nevide

```
data NonEmpty a = a :| [a]      deriving (Eq, Ord)
instance Semigroup (NonEmpty a) where
    (a :| as) <>> (b :| bs) = a :| (as ++ b : bs)
```

Concatenare pentru semigrupuri

```
sconcat :: Semigroup a => NonEmpty a -> a
sconcat (a :| as) = go a as
```

where

```
go a [] = a
go a (b : bs) = a <>> go b bs
```

```
Prelude>sconcat $ (Sum 1) :| [(Sum 2),(Sum 3)]
Sum {getSum = 6}
```

Foldable

din nou foldr

foldr pe liste

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f i [] = i
foldr f i (x:xs) = f x (foldr f i xs)
```

Problema: să generalizăm **foldr** la alte structuri recursive.

Exemplu: arbori binari

```
data BinaryTree a =
    Leaf a
  | Node (BinaryTree a) (BinaryTree a)
deriving Show
```

Cum definim "**foldr**" înlocuind listele cu date de tip **BinaryTree** ?

"foldr" folosind BinaryTree

```
data BinaryTree a =  
    Leaf a  
    | Node (BinaryTree a) (BinaryTree a)  
deriving Show
```

```
foldTree :: (a -> b -> b) -> b -> BinaryTree a -> b
```

```
foldTree f i (Leaf x) = f x i
```

```
foldTree f i (Node l r) = foldTree f (foldTree f i r) l
```

```
myTree = Node (Node (Leaf 1)(Leaf 2))(Node (Leaf 3)(  
    Leaf 4))
```

```
Prelude> foldTree (+) 0 myTree
```

10

clasa Foldable

Data.Foldable

```
class Foldable t where
    fold      :: Monoid m => t m -> m
    foldMap   :: Monoid m => (a -> m) -> t a -> m
    foldr     :: (a -> b -> b) -> b -> t a -> b

    fold = foldMap id
    ...

```

Observații:

- definiția minimală completă conține fie **foldMap**, fie **foldr**
- **foldMap** și **foldr** pot fi definite una prin cealaltă
- pentru a crea o instanță este suficient să definim una dintre **foldMap** și **foldr**, cealaltă va fi automat accesibilă

Foldable cu foldr

```
instance Foldable BinaryTree where
    foldr = foldTree

tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
treeS = Node (Node(Leaf "1")(Leaf "2"))
           (Node (Leaf "3")(Leaf "4"))
```

```
Prelude> foldr (+) 0 tree1
10
Prelude> foldr (++) [] treeS
"1234"
```

clasa Foldable

Data.Foldable

```
class Foldable t where
    fold      :: Monoid m => t m -> m
    foldMap   :: Monoid m => (a -> m) -> t a -> m
    foldr     :: (a -> b -> b) -> b -> t a -> b

    fold = foldMap id
    ...

```

```
instance Foldable BinaryTree where
    foldr = foldTree
```

Observație: în definiția clasei **Foldable**, variabila de tip t nu reprezintă un tip concret ([a], Sum a), ci un **constructor de tip** (BinaryTree)

Foldable cu foldr

Avem definite automat **foldMap** și alte funcții precum: **foldl**, **foldr'**, **foldr1**, ...

```
Prelude> foldl (++) [] treeS
"1234"
Prelude> foldl (+) 0 treeI
10
Prelude> maximum treeI
4
Prelude> foldMap Sum treeI
Sum {getSum = 10}
Prelude> foldMap id treeS
"1234"
```

foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

```
newtype Sum a = Sum { getSum :: a }
                  deriving (Eq, Show)
```

```
instance Num a => Semigroup (Sum a) where
    Sum x <>> Sum y = Sum (x + y)
```

```
instance Num a => Monoid (Sum a) where
    mempty = Sum 0
```

```
tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
```

```
Prelude> foldMap Sum tree1    -- Sum :: a -> Sum a
Sum {getSum = 10}
```

sum cu foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

```
newtype Sum a = Sum { getSum :: a }
```

```
          deriving (Eq, Show)
```

```
instance Num a => Semigroup (Sum a) where
```

```
    Sum x <>> Sum y = Sum (x + y)
```

```
instance Num a => Monoid (Sum a) where
```

```
    mempty = Sum 0
```

```
sum as = getSum $ foldMap Sum as
```

```
-- sum = getSum . (foldMap Sum)
```

```
tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
```

```
Prelude> foldMap Sum tree1 -- Sum :: a -> Sum a
```

```
Sum {getSum = 10}
```

```
Prelude> sum tree1
```

product cu foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m
```

```
newtype Product a = Product { getProduct :: a }
    deriving (Eq, Show)
instance Num a => Semigroup (Product a) where
    Product x <>> Product y = Product (x * y)
instance Num a => Monoid (Product a) where
    mempty = Product 1
```

```
product as = getProduct $ foldMap Product as
-- product = getProduct . (foldMap Product)
```

```
tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
```

```
Prelude> foldMap Product tree1
```

```
Product {getProduct = 24}
```

```
Prelude> product tree1
```

elem cu foldMap

```
foldMap :: Monoid m => (a -> m) -> t a -> m
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Show)
instance Semigroup Any where
    Any x <>> Any y = Any (x || y)
instance Monoid Any where
    mempty = Any False

any as = getAny $ foldMap Any as
-- any = getAny . (foldMap Any)
elem e = getAny . (foldMap (Any . (== e)))
```

```
tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
Prelude> foldMap (Any . (== 1)) tree1
Any {getAny = True}
Prelude> elem 1 tree1
True
```

foldMap folosind **foldr**

Cum definim **foldMap** folosind **foldr**?

```
foldr    :: (a -> b -> b) -> b -> t a -> b  
foldMap   :: Monoid m => (a -> m) -> t a -> m
```

```
foldMap f tr = foldr foo i tr      -- f :: a -> m  
  where foo = ???      -- foo :: (a -> m -> m)  
        i = mempty
```

foldMap folosind **foldr**

Cum definim **foldMap** folosind **foldr**?

foldr :: (a -> b -> b) -> b -> t a -> b

foldMap :: Monoid m => (a -> m) -> t a -> m

```
foldMap f tr = foldr foo i tr      -- f :: a -> m
    where foo = ???      -- foo :: (a -> m -> m)
          i = mempty
```

```
foo = \x acc -> f x <> acc
     = \x acc -> (<>) (f x) acc
     = \x -> (<>) $ f x
     = \x -> ((<>) . f)
     = (<>) . f
```

foldMap f = foldr ((<>) . f) mempty

Foldable cu foldMap

```
instance Foldable BinaryTree where
    foldMap f (Leaf x) = f x
    foldMap f (Node l r) = foldMap f l <>> foldMap f r

tree1 = Node(Node(Leaf 1)(Leaf 2))(Node (Leaf 3)(Leaf 4))
treeS = Node (Node(Leaf "1")(Leaf "2"))
          (Node (Leaf "3")(Leaf "4"))
```

Avem definite automat **foldr** și alte funcții precum: **foldl**,
foldr',**foldr1**,...

```
Prelude> foldr (++) [] treeS
```

```
"1234"
```

```
Prelude> foldl (+) 0 tree1
```

```
10
```

foldr folosind foldMap - facultativ

Cum definim **foldr** folosind **foldMap**?

foldr :: ($a \rightarrow b \rightarrow b$) $\rightarrow b \rightarrow t\ a \rightarrow b$

foldMap :: Monoid m \Rightarrow ($a \rightarrow m$) $\rightarrow t\ a \rightarrow m$

Idee

foldr :: ($a \rightarrow (b \rightarrow b)$) $\rightarrow b \rightarrow t\ a \rightarrow b$

- pentru fiecare element de tip **a** din **t a** se crează o funcție de tip **(b->b)**

*obținem, de exemplu, o lista de funcții sau
un arbore care are ca frunze funcții*

- folosim faptul ca **(b->b)** este instanță a lui **Monoid** și aplicăm **foldMap**

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZv37w>

Seria 24: <https://questionpro.com/t/AT4qgZv68m>

Seria 25: <https://questionpro.com/t/AT4qgZv37u>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C10

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monade - privire de ansamblu

Despre intenție și acțiune

[1] S. Peyton-Jones, Tackling the Awkward Squad: ...

- [1] A purely functional program implements a function; it has no side effect.
- [1] Yet the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent, ...

Exemplu

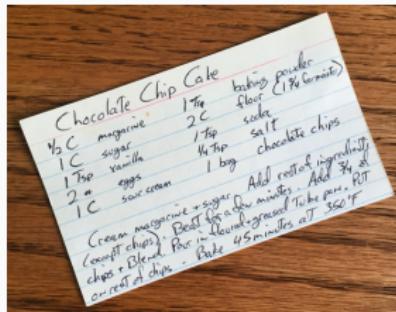
```
putChar :: Char -> IO ()  
Prelude> putChar '!'
```

reprezintă o comandă care, **dacă va fi executată**, va afișa un semn de exclamare.

Mind-Body Problem - Rețetă vs Prăjitură



c :: Cake



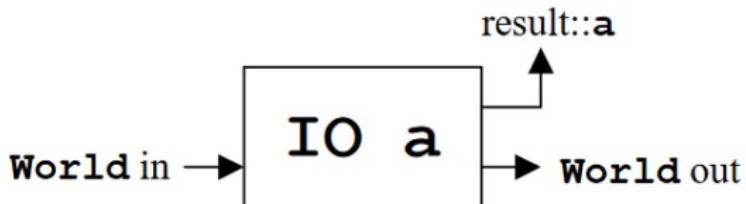
r :: Recipe Cake

IO este o rețetă care produce o valoare de tip a.

Motorul care citeste si executa instructiunile IO se numeste **Haskell Runtime System** (RTS). Acest sistem reprezinta legatura dintre programul scris si mediul în care va fi executat, împreuna cu toate efectele si particularitatile acestuia.

Comenzi în Haskell

```
type IO a = RealWorld -> (a, RealWorld)
```



S. Peyton-Jones, Tackling the Awkward Squad: ...

Ce este o monadă?

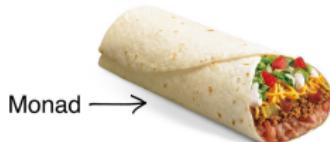
Există multe răspunsuri, variind între

- O monadă este o clasă de tipuri în Haskell.
- "All told, a monad in X is just a monoid in the category of endofunctors in X , with product \times replaced by composition of endofunctors and unit set by the identity endofunctor."

Saunders Mac Lane, Categories for the Working Mathematician, 1998.

- O monadă este un burrito.

<https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/>



Monad →

<https://twitter.com/monadburritos>

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

știind x , obținem **direct** y

- Funcție îmbogățită: $x \mapsto$



știind x , putem să **extragem** y și producem un **efect**

Referințe:

<https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/>

<https://bartoszmilewski.com/2016/11/30/monads-and-effects/>

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



Exemplu

Folosind tipul **Maybe** a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f x = if x < 0 then Nothing else (Just x+1)
```

Cum putem calcula $f \cdot f$?

Functii imbogatite si efecte

Functie imbogatita: $x \mapsto$



Exemplu

Folosind un tip care are ca efect un mesaj

```
newtype Writer log a = Writer {runWriter :: (a, log)}
```



```
f :: Int -> Writer String Int
f x = if x < 0 then (Writer (-x, "negativ"))
      else (Writer (x, "pozitiv"))
```

Cum putem calcula f.f și să concatenăm mesajele?

Clasa de tipuri Monad

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

În Haskell, monada este o clasă de tipuri!

Clasa **Monad** este o extensie a clasei **Applicative**!

- $m a$ este tipul **comenzilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $>>=$ este operația de „secvențiere” a comenzilor

Proprietățile monadelor

Asociativitate și element neutru

Operația $>=$ este asociativă și are element neutru **return**

- Element neutru (la dreapta):

$$(\mathbf{return} \ x) \ >= \ g = g \ x$$

- Element neutru (la stânga):

$$x \ >= \ \mathbf{return} = x$$

- Asociativitate:

$$(f \ >= \ g) \ >= \ h = f \ >= (\lambda x \rightarrow (g \ x \ >= \ h))$$

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do	
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest	<code>binding' :: IO ()</code> <code>binding' =</code> <code>getLine >>= putStrLn</code>
$e >>= _ \rightarrow \text{rest}$	e rest	<code>binding :: IO ()</code> <code>binding = do</code>
$e >> \text{rest}$	e rest	<code>name <- getLine</code> <code>putStrLn name</code>

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do	De exemplu
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest	$e1 >>= \lambda x1 \rightarrow e2 >> e3$
$e >>= _ \rightarrow \text{rest}$	e rest	devine
$e >> \text{rest}$	e rest	do $x1 \leftarrow e1$ e2 e3

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

De exemplu

$e1\ >>= \ \lambda x1 \rightarrow e2\ >>= \ \lambda x2 \rightarrow e3\ >>= \ \lambda _ \rightarrow e4\ >>= \ \lambda x4 \rightarrow e5$

devine

do

$x1 \leftarrow e1$

$x2 \leftarrow e2$

$e3$

$x4 \leftarrow e4$

$e5$

Functor și Applicative definiți cu return și >>=

```
instance Monad M where
```

```
    return a = ...
```

```
    ma >>= k = ...
```

```
instance Applicative M where
```

```
    pure = return
```

```
    mf <*> ma = do
```

```
        f <- mf
```

```
        a <- ma
```

```
        return (f a)
```

```
-- mf >>= (\f -> ma >>= (\a -> return (f a)))
```

```
instance Functor M where
```

```
    fmap f ma = pure f <*> ma
```

```
-- ma >>= \a -> return (f a)
```

```
-- ma >>= (return . f)
```

Exemple de efecte laterale

I/O	Monada IO
Parțialitate	Monada Maybe
Excepții	Monada Either
Nedeterminism	Monada <code>[]</code> (listă)
Logging	Monada Writer
Stare	Monada State
Memorie read-only	Monada Reader

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C10

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monade - privire de ansamblu

Despre intenție și acțiune

[1] S. Peyton-Jones, Tackling the Awkward Squad: ...

- [1] A purely functional program implements a function; it has no side effect.
- [1] Yet the ultimate purpose of running a program is invariably to cause some side effect: a changed file, some new pixels on the screen, a message sent, ...

Exemplu

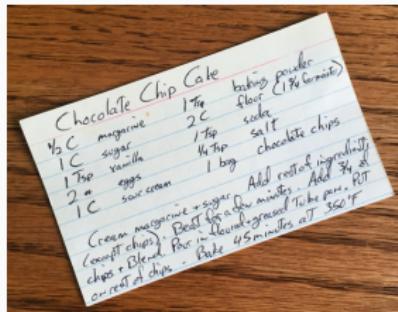
```
putChar :: Char -> IO ()  
Prelude> putChar '!'
```

reprezintă o comandă care, **dacă va fi executată**, va afișa un semn de exclamare.

Mind-Body Problem - Rețetă vs Prăjitură



c :: Cake



r :: Recipe Cake

IO este o rețetă care produce o valoare de tip a.

Motorul care citeste si executa instructiunile IO se numeste **Haskell Runtime System** (RTS). Acest sistem reprezinta legatura dintre programul scris si mediul în care va fi executat, împreuna cu toate efectele si particularitatile acestuia.

Ce este o monadă?

Există multe răspunsuri, variind între

- O monadă este o clasă de tipuri în Haskell.
- "All told, a monad in X is just a monoid in the category of endofunctors in X , with product \times replaced by composition of endofunctors and unit set by the identity endofunctor."

Saunders Mac Lane, Categories for the Working Mathematician, 1998.

- O monadă este un burrito.

<https://byorgey.wordpress.com/2009/01/12/abstraction-intuition-and-the-monad-tutorial-fallacy/>



Monad →

<https://twitter.com/monadburritos>

Funcții îmbogățite și efecte

- Funcție simplă: $x \mapsto y$

știind x , obținem **direct** y

- Funcție îmbogățită: $x \mapsto$



știind x , putem să **extragem** y și producem un **efect**

Referințe:

<https://bartoszmilewski.com/2016/11/21/monads-programmers-definition/>

<https://bartoszmilewski.com/2016/11/30/monads-and-effects/>

Funcții îmbogățite și efecte

Funcție îmbogățită: $x \mapsto$



Exemplu

Folosind tipul **Maybe** a

```
data Maybe a = Nothing | Just a
f :: Int -> Maybe Int
f x = if x < 0 then Nothing else (Just x+1)
```

Cum putem calcula $f \cdot f$?

Functii imbogatite si efecte

Functie imbogatita: $x \mapsto$



Exemplu

Folosind un tip care are ca efect un mesaj

```
newtype Writer log a = Writer {runWriter :: (a, log)}
```



```
f :: Int -> Writer String Int
f x = if x < 0 then (Writer (-x, "negativ"))
      else (Writer (x, "pozitiv"))
```

Cum putem calcula f.f și să concatenăm mesajele?

Clasa de tipuri Monad

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

În Haskell, monada este o clasă de tipuri!

Clasa **Monad** este o extensie a clasei **Applicative**!

- $m a$ este tipul **comenzilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $>>=$ este operația de „secvențiere” a comenzilor

Proprietățile monadelor

Asociativitate și element neutru

Operația $>=$ este asociativă și are element neutru **return**

- Element neutru (la dreapta):

$$(\mathbf{return} \ x) \ >= \ g = g \ x$$

- Element neutru (la stânga):

$$x \ >= \ \mathbf{return} = x$$

- Asociativitate:

$$(f \ >= \ g) \ >= \ h = f \ >= (\lambda x \rightarrow (g \ x \ >= \ h))$$

De ce nu este suficient fmap?

Exemplu: **putStrLn <\$> getLine**

```
<$> :: Functor f => (a -> b) -> f a -> f b  
getLine :: IO String  
putStrLn :: String -> IO ()  
-- in exemplul nostru, b devine IO ()
```

```
-- [1] [2] [3]  
putStrLn <$> getLine :: IO (IO ())
```

[1] **IO** din exterior reprezinta efectul pe care **getLine** trebuie sa il produca pentru a obtine un **String** introdus de utilizator

[2] **IO** din interior reprezinta efectul care s-ar produce daca **putStrLn** a fost evaluat

[3] () este tipul unitate intors de **putStrLn**

De ce nu este suficient fmap?

```
putStrLn <$> getLine :: IO (IO ())
```

Trebuie sa unim efectele lui **getLine** si **putStrLn** intr-un singur efect **IO!**

```
import Control.Monad (join)
join :: Monad m => m (m a) -> m a
join $ putStrLn <$> getLine
```

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do	
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest	<code>binding' :: IO ()</code> <code>binding' =</code> <code>getLine >>= putStrLn</code>
$e >>= _ \rightarrow \text{rest}$	e rest	<code>binding :: IO ()</code> <code>binding = do</code> <code>name <- getLine</code> <code>putStrLn name</code>
$e >> \text{rest}$	e rest	

Notația do pentru monade

```
twoBinds' :: IO ()
twoBinds' =
    putStrLn "name pls:" >>
    getLine >>=
        \name ->
            putStrLn "age pls:" >>
            getLine >>=
                \age ->
                    putStrLn ("y helo thar: "
                        ++ name ++ " who is: "
                        ++ age ++ " years old.")
```

Notația do pentru monade

```
twoBinds :: IO ()
twoBinds = do
    putStrLn "name pls:"
    name <- getLine

    putStrLn "age pls:"
    age <- getLine

    putStrLn ("y helo thar: "
              ++ name ++ " who is: "
              ++ age ++ " years old.")
```

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do	De exemplu
$e >>= \lambda x \rightarrow \text{rest}$	$x <- e$ rest	$e1 >>= \lambda x1 \rightarrow e2 >> e3$
$e >>= _ \rightarrow \text{rest}$	e rest	devine
$e >> \text{rest}$	e rest	

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do	De exemplu
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest	$e1 >>= \lambda x1 \rightarrow e2 >> e3$
$e >>= _ \rightarrow \text{rest}$	e rest	devine
$e >> \text{rest}$	e rest	do $x1 \leftarrow e1$ e2 e3

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

De exemplu

$e1\ >>= \ \lambda x1 \rightarrow e2\ >>= \ \lambda x2 \rightarrow e3\ >>= \ \lambda _ \rightarrow e4\ >>= \ \lambda x4 \rightarrow e5$

devine

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

De exemplu

$e1\ >>= \ \lambda x1 \rightarrow e2\ >>= \ \lambda x2 \rightarrow e3\ >>= \ \lambda _ \rightarrow e4\ >>= \ \lambda x4 \rightarrow e5$

devine

do

$x1 \leftarrow e1$

$x2 \leftarrow e2$

$e3$

$x4 \leftarrow e4$

$e5$

Functor și Applicative definiți cu return și >>=

```
instance Monad M where
```

```
    return a = ...
```

```
    ma >>= k = ...
```

```
instance Applicative M where
```

```
    pure = return
```

```
    mf <*> ma = do
```

```
        f <- mf
```

```
        a <- ma
```

```
        return (f a)
```

```
-- mf >>= (\f -> ma >>= (\a -> return (f a)))
```

```
instance Functor M where
```

```
    fmap f ma = pure f <*> ma
```

```
-- ma >>= \a -> return (f a)
```

```
-- ma >>= (return . f)
```

Exemple de efecte laterale

I/O	Monada IO
Parțialitate	Monada Maybe
Excepții	Monada Either
Nedeterminism	Monada <code>[]</code> (listă)
Logging	Monada Writer
Stare	Monada State
Memorie read-only	Monada Reader

Monada listă

Monada pentru liste – nedeterminism

O computatie care intoarce un rezultat nedeterminist nu este o functie pura, dar poate fi transformata intr-o functie pura transformand rezultattul sau din tipul a in tipul [a].

In esenta, construim o functie care returneaza toate rezultatele posibile in acelasi timp.

Monada pentru liste – nedeterminism

Exemplu:

- Un joc de sah care evalueaza mutarile viitoare
- Trebuie sa anticipeze mutarile unui adversar nedeterminist
- Doar o mutare a adversarului se va concretiza, dar toate mutarile posibile trebuie luate in calcul in planificarea unei strategii
- O operatie de baza intr-un astfel de program este `move`: ia o stare a tablei de sah si returneaza o noua stare a tablei dupa ce o miscare a fost efectuata
- Dar ce ar trebui sa returneze `move`? Sunt multe mutari posibile in fiecare situatie, rezultatul fiind **nedeterminist**
- Ca sa putem compune astfel de mutari, trebuie sa definim o monada.

Instanta de Monade pentru liste

Functiile din clasa **Monad** specialized pentru liste:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
return :: a -> [a]
```

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
```

Monada pentru liste – exemplu

```
twiceWhenEven :: [Integer] -> [Integer]
twiceWhenEven xs = do
    x <- xs
    if even x
        then [x*x, x*x]
        else [x*x]
```

Vacanta placuta!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C11- Seriile 23 si 25

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monade

Clasa de tipuri Monad

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

În Haskell, monada este o clasă de tipuri!

Clasa **Monad** este o extensie a clasei **Applicative**!

- $m a$ este tipul **comenzilor** care produc rezultate de tip a (și au efecte laterale)
- $a \rightarrow m b$ este tipul **continuărilor** / a funcțiilor cu efecte laterale
- $>>=$ este operația de „secvențiere” a comenzilor

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do	
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest	<code>binding' :: IO ()</code> <code>binding' =</code> <code>getLine >>= putStrLn</code>
$e >>= _ \rightarrow \text{rest}$	e rest	<code>binding :: IO ()</code> <code>binding = do</code>
$e >> \text{rest}$	e rest	<code>name <- getLine</code> <code>putStrLn name</code>

Notația do pentru monade

```
twoBinds' :: IO ()
twoBinds' =
    putStrLn "name pls: " >>
    getLine >>=
        \name ->
            putStrLn "age pls :" >>
            getLine >>=
                \age ->
                    putStrLn ("hello: "
                        ++ name ++ " who is: "
                        ++ age ++ " years old.")
```

Notația do pentru monade

```
twoBinds :: IO ()
twoBinds = do
    putStrLn "name pls: "
    name <- getLine

    putStrLn "age pls: "
    age <- getLine

    putStrLn ("hello: "
              ++ name ++ " who is: "
              ++ age ++ " years old.")
```

Notația do pentru monade

$(>>=)$:: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
 $(>>)$:: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do	De exemplu
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest	$e1 >>= \lambda x1 \rightarrow e2 >> e3$
$e >>= _ \rightarrow \text{rest}$	e rest	devine
$e >> \text{rest}$	e rest	

Notația do pentru monade

$(>>=)$:: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
 $(>>)$:: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do	De exemplu
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest	$e1 >>= \lambda x1 \rightarrow e2 >> e3$
$e >>= _ \rightarrow \text{rest}$	e rest	devine do
$e >> \text{rest}$	e rest	$x1 \leftarrow e1$ e2 e3

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

De exemplu

$e1\ >>= \ \lambda x1 \rightarrow e2\ >>= \ \lambda x2 \rightarrow e3\ >>= \ \lambda _ \rightarrow e4\ >>= \ \lambda x4 \rightarrow e5$

devine

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$
($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

De exemplu

$e1\ >>= \ x1 \rightarrow e2\ >>= \ x2 \rightarrow e3\ >>= \ _ \rightarrow e4\ >>= \ x4 \rightarrow e5$

devine

do

$x1 \leftarrow e1$

$x2 \leftarrow e2$

$e3$

$x4 \leftarrow e4$

$e5$

Functor și Applicative definiți cu return și >>=

```
instance Monad M where
```

```
    return a = ...
```

```
    ma >>= k = ...
```

```
instance Applicative M where
```

```
    pure = return
```

```
    mf <*> ma = do
```

```
        f <- mf
```

```
        a <- ma
```

```
        return (f a)
```

```
-- mf >>= (\f -> ma >>= (\a -> return (f a)))
```

```
instance Functor M where
```

```
    fmap f ma = pure f <*> ma
```

```
-- ma >>= \a -> return (f a)
```

```
-- ma >>= (return . f)
```

Exemple de efecte laterale

I/O	Monada IO
Parțialitate	Monada Maybe
Excepții	Monada Either
Nedeterminism	Monada <code>[]</code> (listă)
Logging	Monada Writer
Stare	Monada State
Memorie read-only	Monada Reader

Monada listă

Monada pentru liste – nedeterminism

O computatie care intoarce un rezultat nedeterminist nu este o functie pura, dar poate fi transformata intr-o functie pura transformand rezultatul sau din tipul a in tipul $[a]$.

In esenta, construim o functie care returneaza toate rezultatele posibile in acelasi timp.

Monada pentru liste – nedeterminism

Exemplu:

- Un joc de sah care evalueaza mutarile viitoare
- Trebuie sa anticipeze mutarile unui adversar nedeterminist
- Doar o mutare a adversarului se va concretiza, dar toate mutarile posibile trebuie luate in calcul in planificarea unei strategii
- O operatie de baza intr-un astfel de program este `move`: ia o stare a tablei de sah si returneaza o noua stare a tablei dupa ce o miscare a fost efectuata
- Dar ce ar trebui sa returneze `move`? Sunt multe mutari posibile in fiecare situatie, rezultatul fiind **nedeterminist**
- Ca sa putem compune astfel de mutari, trebuie sa definim o monada.

Instanta de Monade pentru liste

Functiile din clasa **Monad** specialized pentru liste:

```
(>>=) :: [a] -> (a -> [b]) -> [b]
return :: a -> [a]
```

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs) -- [vb | va <- ma, vb
                                         <- k va]
```

Monada pentru liste – exemplu

```
twiceWhenEven :: [Integer] -> [Integer]
twiceWhenEven xs = do
    x <- xs
    if even x
        then [x*x, x*x]
        else [x*x]
```

Monada pentru liste – exemplu

```
radical :: Float -> [Float]
radical x
| x >= 0 = [negate (sqrt x), sqrt x]
| x < 0  = []

solEq2 :: Float -> Float -> Float -> [Float]
solEq2 0 0 c = []           -- a * x^2 + b * x + c = 0
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return (negate b + rDelta) / (2 * a)
```

Monada Maybe(a funcțiilor parțiale)

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

```
instance Monad Maybe where
```

```
    return = Just
```

```
    Just va >>= k = k va
```

```
    Nothing >>= _ = Nothing
```

Monada Maybe – exemplu

```
radical :: Float -> Maybe Float
radical x
| x >= 0 = return (sqrt x)
| x < 0 = Nothing
```

```
solEq2 :: Float -> Float -> Float -> Maybe Float
solEq2 0 0 0 = return 0      -- a * x^2 + b * x + c = 0
solEq2 0 0 c = Nothing
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return ((negate b + rDelta) / (2 * a))
```

Monada Either(a excepțiilor)

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

instance Monad (Either err) where
    return = Right
    Right va >>= k = k va
    err       >>= _ = err
    -- Left verr >>=_ = Left verr
```

Monada Either – exemplu

```
radical :: Float -> Either String Float
radical x
| x >= 0 = return (sqrt x)
| x < 0  = Left "radical: argument negativ"

solEq2 :: Float -> Float -> Float -> Either String
        Float
solEq2 0 0 0 = return 0      -- a * x^2 + b * x + c = 0
solEq2 0 0 c = Left "Nu are solutii"
solEq2 0 b c = return ((negate c) / b)
solEq2 a b c = do
    rDelta <- radical (b * b - 4 * a * c)
    return ((negate b + rDelta) / (2 * a))
```

Monada Writer (variantă simplificată)

```
newtype Writer log a = Writer {runWriter :: (a, log)}  
-- a este parametru de tip  
  
instance Monad (Writer String) where  
    return va = Writer (va, "")  
    ma >>= k =  
        let (va, log1) = runWriter ma  
            (vb, log2) = runWriter (k va)  
        in Writer (vb, log1 ++ log2)
```

Monada Writer - Exemplu logging

```
newtype Writer log a = Writer {runWriter :: (a, log)}
```



```
tell :: log -> Writer log ()
```

```
tell msg = Writer ((), msg)
```



```
logIncrement :: Int -> Writer String Int
```

```
logIncrement x = do
    tell ("increment: " ++ show x ++ "\n")
    return (x + 1)
```

```
logIncrement2 :: Int -> Writer String Int
```

```
logIncrement2 x = do
    y <- logIncrement x
    logIncrement y
```

```
Main> runWriter (logIncrement2 13)
(15,"increment: 13\nincrement: 14\n")
```

Vacanta placuta!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C11- Seriile 23 si 25

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Recap

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

class Functor m => Applicative m where
    pure :: a -> m a
    (<*>) :: m (a -> b) -> m a -> m b

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

Functor și Applicative definiți cu return și >>=

```
instance Monad M where
```

```
    return a = ...
```

```
    ma >>= k = ...
```

```
instance Applicative M where
```

```
    pure = return
```

```
    mf <*> ma = do
```

```
        f <- mf
```

```
        a <- ma
```

```
        return (f a)
```

```
instance Functor M where
```

```
    fmap f ma = pure f <*> ma
```

Notația do pentru monade

($>>=$) :: $m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ b$

($>>$) :: $m\ a \rightarrow m\ b \rightarrow m\ b$

Notația cu operatori	Notația do
$e >>= \lambda x \rightarrow \text{rest}$	$x \leftarrow e$ rest
$e >>= _ \rightarrow \text{rest}$	e rest
$e >> \text{rest}$	e rest

binding' :: **IO** ()
binding' =
getLine >>= **putStrLn**

binding :: **IO** ()
binding = **do**
 name <- **getLine**
 putStrLn name

Monada Writer (variantă simplificată)

```
newtype Writer log a = Writer {runWriter :: (a, log)}
-- a este parametru de tip

tell :: log -> Writer log ()
tell msg = Writer ((), msg)

instance Monad (Writer String) where
    return va = Writer (va, "")
    ma >>= k = let (va, log1) = runWriter ma
               (vb, log2) = runWriter (k va)
               in Writer (vb, log1 ++ log2)
```

Monada Writer (varianta generalizată)

```
class Semigroup a where
  (<>) :: a -> a -> a

class Semigroup a => Monoid a where
  mempty :: a
  mappend :: a -> a -> a
  mappend = (<>)

newtype Writer log a = Writer {runWriter :: (a, log)}

instance Monoid log => Monad (Writer log) where
  return a = Writer (a, mempty)
  ma >>= k = let (va, log1) = runWriter ma
              (vb, log2) = runWriter (k va)
            in Writer (vb, log1 `mappend` log2)
```

Monada Reader(stare nemodificabilă)

```
newtype Reader env a = Reader {runReader :: env -> a}

ask :: Reader env env
ask = Reader id

instance Monad (Reader env) where
    return x = Reader (\_ -> x)
    -- return x = Reader (const x)
    ma >>= k = Reader f
        where
            f env = let va = runReader ma env
                    in runReader (k va) env
```

Monada Reader- exemplu: mediu de evaluare

```
newtype Reader env a = Reader {runReader :: env -> a}
data Prop = Var String | Prop :&: Prop
type Env = [(String, Bool)]

var :: String -> Reader Env Bool
var x = do
    env <- ask
    return $ fromMaybe False (lookup x env)

eval :: Prop -> Reader Env Bool
eval (Var x) = var x
eval (p1 :&: p2) = do
    b1 <- eval p1
    b2 <- eval p2
    return (b1 && b2)

runEval prop env = runReader (eval prop) env
```

Monada State

```
newtype State state a = State{runState :: state ->(a,
    state)}
```



```
instance Monad (State state) where
    return va = State (\s -> (va, s))
    -- return a = State f where f s = (a,s)
```



```
ma >>= k = State \$ \s -> let
    (va, news) = runState ma s
    State h = k va
    in (h news)
```



```
-- ma :: State state a
-- runState ma :: state -> (a, state)
-- k :: a -> State state b
-- h :: state -> (b, state)
-- ma >>= k :: State state b
```

Monada State

```
newtype State state a = State{runState :: state ->(a,  
state)}
```



```
instance Monad (State state) where  
    return va = State (\s -> (va, s))  
    ma >>= k = State \$ \s -> let  
        (va, news) = runState ma s  
        State h = k va  
    in (h news)
```

Functii ajutatoare:

```
get :: State state state  
get = State (\s -> (s, s))
```

```
modify :: (state -> state) -> State state ()  
modify f = State (\s -> (((), f s)))
```

Monada State - exemplu "random"

```
newtype State state a = State{runState :: state ->(a,  
state)}
```

```
cMULTIPLIER, cINCREMENT :: Word32 -- 32-bit unsigned  
integer type
```

```
cMULTIPLIER = 1664525 ; cINCREMENT = 1013904223
```

```
rnd, rnd2 :: State Word32 Word32  
rnd = do  
    modify (\seed -> cMULTIPLIER * seed + cINCREMENT)  
    get
```

```
rnd2 = do  
    r1 <- rnd  
    r2 <- rnd  
return (r1 + r2)
```

```
-- runState rnd2 0 = (2210339985,1196435762)
```

Pe data viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C13

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Monada IO

Monada IO

IO a corespunde comenzilor care produc rezultate de tip a.

Exemplu: citește un caracter

IO Char corespunde comenzilor care produc rezultate de tip **Char**

`getChar :: IO Char`

Poate să citească de la tastatura o literă.

IO () corespunde comenzilor care nu produc rezultate

- () este tipul unitate care conține doar valoarea ()

Combinarea comenziilor cu valori

`(>>=) :: IO a -> (a -> IO b) -> IO b`

Exemplu

`getChar :: IO Char`

`putChar :: Char -> IO ()`

`getChar >>= \x -> putChar (toUpper x)`

Poate sa citeasca de la tastatura o litera si sa o afiseze pe ecran, trasformata in litera mare.

Operatorul de legare / bind

$$(>=) :: \text{IO } a \rightarrow (a \rightarrow \text{IO } b) \rightarrow \text{IO } b$$

- Dacă fiind o comandă care produce o valoare de tip a
 $m :: \text{IO } a$
- Date fiind o funcție care pentru o valoare de tip a se evaluează la o comandă de tip b
 $k :: a \rightarrow \text{IO } b$
- Atunci
 $m >= k :: \text{IO } b$
este comanda care, dacă se va executa:
 - Mai întâi efectuează m, obținând valoarea x de tip a
 - Apoi efectuează comanda k x obținând o valoare y de tip b
 - Produce y ca rezultat al comenzi

Exemplu: citește o linie

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

Exemplu: citește o linie în notatia "do"

Urmatoarele sunt echivalente:

myLine :: IO String	myLineDo :: IO String
myLine = getChar >>= \x ->	myLineDo = do
if x == '\n' then	x <- getChar
return []	if x == '\n' then
else	return []
myLine >>= \xs ->	else do
return (x:xs)	xs <- myLineDo
	return (x:xs)

Exemplu: de la intrare la ieșire

```
echo :: IO ()  
echo = getLine >>= \line ->  
    if line == "" then  
        return ()  
    else  
        putStrLn (map toUpper line) >>  
        echo
```

```
*C13> echo  
One line  
ONE LINE  
And, another line !  
AND, ANOTHER LINE !  
...
```

Exemplu: de la intrare la ieșire în notatia "do"

```
echo :: IO ()  
echo = getLine >>= \line ->  
    if line == "" then  
        return ()  
    else  
        putStrLn (map toUpper line) >>  
        echo
```

Echivalent cu

```
echoDo :: IO ()  
echoDo = do  
    line <- getLine  
    if line == "" then  
        return ()  
    else do  
        putStrLn (map toUpper line)  
        echoDo
```

Exemplu: citirea si afisarea unui numar

```
myNumber = putStrLn "Please enter a number: " >>
    readLn >= \n ->
    let m = n + 1 in
    print m -- putStrLn (show m)
```

```
myNumberDo = do
    putStrLn "Please enter a number: "
    n <- readLn
    let m = n + 1
    print m
```

Examen

Notare

- Nota finală: 1 (oficiu) + parțial + examen
- Restanță: 1 (oficiu) + examen
(parțialul nu se ia în calcul la restanță)

Condiție de promovabilitate

- cel puțin **5** > 4.99

Activitate laborator

- La sugestia instructorilor de la laborator, se poate nota activitatea în plus față de cerințele obișnuite.
- Maxim 1 punct (bonus la nota finală)

Punctele bonus nu se pot folosi în condiția de promovabilitate!

Punctele bonus nu se pot folosi în restanță!

Examen final

- valorează **6 puncte** din nota finală
- **1 februarie**
 - seria 23 + 251 - ora 10:00 [o ora]
 - seria 24 + 252 - ora 12:00 [o ora]
- pe calculatoarele din laboratoarele facultatii
- vom posta pe Teams o distributie in laboratoare
- acoperă toată materia
- materiale ajutătoare: suporturile de curs si de laborator
- fără internet
- va conține exerciții asemănătoare cu cele de la laborator
[vezi **model de examen**]

Concluzii

Conținutul cursului de PF 2022

- Elemente de baza
- Functii
- Liste
- Currying
- Operatori. Secțiuni
- Functii de nivel inalt
- Tipuri de date algebrice
- Clase de tipuri
- Functor
- Semigroup
- Monoid
- Foldable
- Applicative
- Monad

Haskell - best for

Application Domains

- Compilers
- Server-side web programming
- Scripting / Command-line applications
- ...

Common Programming Needs

- Maintenance
- Single-machine Concurrency
- Types / Type-driven development
- Parsing / Pretty-printing
- Domain-specific languages (DSLs)
- ...

More to explore

- Testing; Property-Based Testing
- Random Numbers
- Monad Transformers
- Parallel and Concurrent Programming
- Dependent Types

Feedback

Seria 23: <https://questionpro.com/t/AT4qgZwUpF>

Seria 24: <https://questionpro.com/t/AT4NiZwUI9>

Seria 25: <https://questionpro.com/t/AT4qgZwUpH>

Mulțumim că ați participat la acest curs!

Multă baftă la examen!