

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C01 - Organizare. Elemente de bază

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Organizare

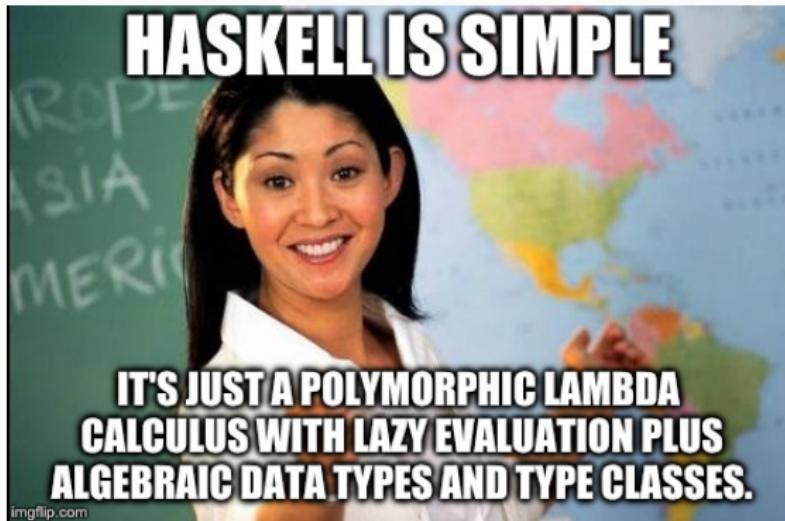
Instructori

- Curs
 - Claudia Chiriță (seria 23)
 - Denisa Diaconescu (seriile 24,25)
- Laborator
 - Andrei Burdușa
 - Horatiu Cheval
 - Ioan Ionescu
 - Alina Georgescu
 - Alex Oltean
 - George-Mihai Radu
 - Stelian Stoian

Suport curs

- Suporturi de curs și laborator:
<https://tinyurl.com/PF2024-Drive>
- Canal Teams:
<https://tinyurl.com/PF2024-MTeams>

Prezență la curs sau la laboratoare nu este obligatorie,
dar extrem de încurajată!



Evaluare

Notare

- **Sesiune:** 1 (oficiu) + parțial + examen
- **Restanță 1:** la alegere una din variantele de mai jos
 - 1 (oficiu) + parțial + examen (dacă există notă la parțial și vreți să o păstrati)
 - 1 (oficiu) + examen (altfel)
- **Restanță 2:** 1 (oficiu) + examen

Condiție de promovabilitate

- **cel puțin 5 > 4.99**

Partial

- valorează **3 puncte** din nota finală
- durează **40 min**
- în **săptămâna 9 în cadrul cursului** (25-29 noiembrie)
- nu este obligatoriu și nu se poate reface
- va conține **15 întrebări grilă**, asemănătoare cu cele din curs
- materiale ajutătoare: suporturile de curs și de laborator, notițe proprii
- fără aparate electronice (laptop, tabletă, telefon etc.)

Examen final

- valorează **6 puncte** din nota finală
- durează **1 oră**
- în sesiune, fizic
- acoperă toată materia
- va conține **exerciții** asemănătoare cu cele de la laborator
- materiale ajutătoare: suporturile de curs și de laborator, notițe proprii
- se poate susține pe laptop propriu sau pe un calculator din facultate

Bonus

- maxim **1 punct** pentru activitatea de la laborator
- nu se poate folosi pentru a satisface condiția de promovabilitate

Nu trăsați, cereti-ne ajutorul!



Programare funcțională în Haskell

- Tipuri, funcții
- 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/>
- Fundație independentă și non-profit ce are ca scop îmbunătățirea experienței cu limbajul Haskell
- Oferă suport pentru biblioteci, tool-uri, educație, cercetare

De ce programare funcțională?

Sondaj

Ce știți despre programarea funcțională?



<https://tinyurl.com/PF2023-C01-Quiz1>

Programare declarativă vs. imperativă – Ce vs. cum

Programare imperativă (Cum)

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

- limbi de programare procedurale
- limbi 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 :-)

- limbi de programare logică
- limbi de interogare a bazelor de date
- limbi 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>



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;  
}
```



A large screenshot of a presentation slide. At the top, there is some text that appears to be part of a larger code block. Below that, the slide contains two versions of a JavaScript function, "multiply". The first version is declarative, using the reduce method. The second version is imperative, using a for loop to iterate through the array and calculate the total. Both versions return the total product.

Agregarea datelor dintr-o colecție (JS)

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

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

The slide features a large metal gear icon on the left and a bunch of bananas on the right. A cartoon Minion is shown at the bottom right, holding a can. The title 'Reasons to be More Declarative' is centered above a bulleted list.

Reasons to be More Declarative

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

A video frame at the bottom left shows a person standing at a podium, likely giving a talk. The video frame has a red border.

A video frame at the bottom right shows a presentation slide with some code snippets. The video frame has a red border.

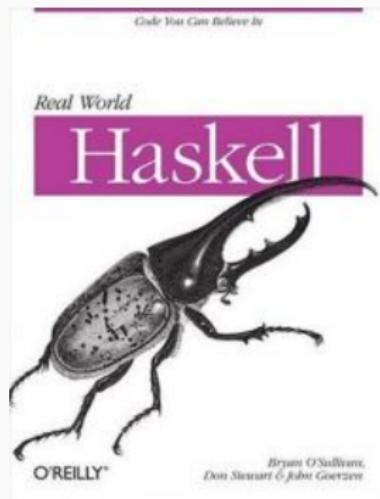
Programare funcțională în Haskell

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.



Programare funcțională în Haskell

Exemplu (Ciurul lui Eratostene)

https://ro.wikipedia.org/wiki/Ciurul_lui_Eratostene

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

Haskell este un limbaj funcțional pur



- Funcțiile sunt *first-class citizens*.
 - Funcțiile sunt folosite ca valori.
 - De exemplu, funcțiile pot fi transmise ca argumente și pot fi returnate de alte funcții.
- Funcțiile sunt pure.
 - Produc aceleasi rezultate pentru aceleasi intrari.
 - O bucată de cod nu poate corupe datele altor bucati de cod.
 - Distincție clară între părțile pure și părțile 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 este lenes și minimalist

- **Lazyness:** orice calcul este 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 greșeli
 - ... rezolvând totuși problema :-)
- multiply = **foldl** (*) 1
doubled = **map** (* 2)
- Oferă suport pentru paralelism și concurență.

Exemplu

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

Exemplu - generalizare

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

Haskell în industrie

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

Haskell e 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:

Typeable Blog Post: 7 Useful Tools Written in Haskell

Serokell Blog Post: Best Haskell open source projects

Serokell Blog Post: Why Fintech Companies Use Haskell

Wasp Blog Post: How to get started with Haskell in 2022

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>

Magia din spate: λ -calcul

În 1929-1932, Alonzo 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.

$t = \quad x$ (variabilă)
| $\lambda x. t$ (abstractizare)
| $t t$ (aplicare)



Magia din spate: λ -calcul

- Independent, în 1935 Alan Turing a introdus mașina Turing.
- În 1936, Turing a argumentat că orice funcție calculabilă peste numere naturale poate fi calculată de o mașină Turing.
- Turing a arătat echivalența celor două modele de calcul.
- Această echivalență a constituit o indicație puternică asupra "universalității" celor două modele (Teza Church-Turing).

Elemente de bază. Primii pași

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 fișierul test.hs conține

x = 1

x = 2

Ce valoare are x?

Variabile

Presupunem că fișierul 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 creează un domeniu de vizibilitate local.

Presupunem că fișierul testlet.hs conține

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

Ce valoare au z și x?

Legarea variabilelor

let .. in ... este o expresie care creează un domeniu de vizibilitate local.

Presupunem că fișierul testlet.hs conține

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

Ce valoare au z și x?

```
-- z = 3  
-- x = 1
```

Legarea variabilelor

let .. in ... este o **expresie** care creează un domeniu de vizibilitate 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 creează un domeniu de vizibilitate 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 creează un domeniu de vizibilitate local.

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

Ce valoare are x?

```
-- x = 12
```

Dar în situația de mai jos, ce valoare are x?

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

Legarea variabilelor

let .. in ... este o expresie care creează un domeniu de vizibilitate local.

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

Ce valoare are x?

```
-- x = 12
```

Dar în situația de mai jos, ce valoare are x?

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

Legarea variabilelor

clauza **... where ...** creează un domeniu de vizibilitate local

$f \ x = g \ x + g \ x + z$

where

$g \ x = 2*x$

$z = x - 1$

Ce valoare are $f \ 1$?

Legarea variabilelor

clauza **... where ...** creează un domeniu de vizibilitate local

$f\ x = g\ x + g\ x + z$

where

$g\ x = 2*x$

$z = x - 1$

Ce valoare are $f\ 1$?

$--\ x = 4$

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!



<https://tinyurl.com/PF2023-C01-Quiz2>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C02 - Tipuri de date

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Tipuri de date

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 și 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 și 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" ]
```

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

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 formalii
- 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)
```

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

În 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

În 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 `_`, `x`, **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]

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 -> 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]
```

Liste

Liste

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

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

Definiție recursivă. O **listă** este

- **vidă**, notată `[]`, sau
- **compusă**, notată `x : xs`, dintr-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> import Data.List
```

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

```
3
```

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

```
'a'
```

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

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

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 $\{x \mid P(x)\}$

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

Putem folosi **let** pentru domeniu de vizibilitate local.

```
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')]
```

Lazy

Lazy: 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.

```
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.

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

Şabloanele sunt **liniare**, adică o variabilă apare cel mult o dată.

Şabloanele în care o variabilă apare de mai multe ori generează 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!



<https://tinyurl.com/PF2023-C02-Quiz1>

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	<code>!!</code>		<code>.</code>
8			<code>^, ^^, **</code>
7	<code>*, /, 'div', 'mod', 'rem', 'quot'</code>		
6	<code>+, -</code>		
5			<code>: , ++</code>
4		<code>==, /=, <, <=, >, >=, 'elem', 'notElem'</code>	
3			<code>&&</code>
2			<code> </code>
1	<code>>>, >>=</code>		
0			<code>\$, \$!, 'seq'</code>

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

$$(++) :: [a] \rightarrow [a] \rightarrow [a]$$

$$[] ++ ys = ys$$

$$(x:xs) ++ ys = x:(xs ++ ys)$$

$$[1 ++ [2 ++ [3 ++ [4 ++ [5]]]] == [1 ++ ([2 ++ ([3 ++ ([4 ++ [5]]))])]$$

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
```

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C03

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

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 folosind şablonane

flip f x y = f y x

- definiția cu lambda expresii

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!



<https://tinyurl.com/PF2023-C03-Quiz1>

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/uncurry și mulțimi

```
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"
```

Quiz time!



<https://tinyurl.com/PF2023-C03-Quizz2>

Procesarea fluxurilor de date: Map, Filter, Fold

Map:

transformarea fiecărui element dintr-o listă

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 xs = [f x | x <- xs]
```

```
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!



<https://tinyurl.com/PF2023-C04-Quiz1>

Filter:

selectarea elementelor dintr-o listă

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 selecteaza 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!



<https://tinyurl.com/PF2023-C04-Quiz2>

Fold:

agregarea elementelor dintr-o listă

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
```

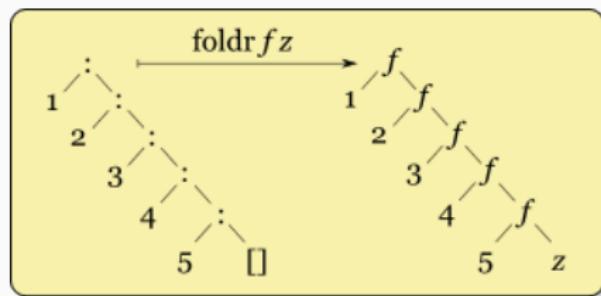
```
Prelude> concat [[1,2,3],[4,5]]
[1,2,3,4,5]
```

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

Funcția 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." ++ []))
```

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C04

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Fold:

agregarea elementelor dintr-o listă

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
```

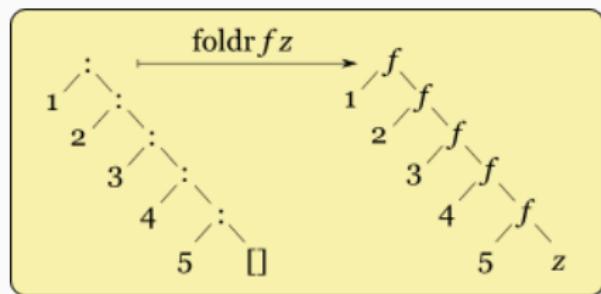
```
Prelude> concat [[1,2,3],[4,5]]
[1,2,3,4,5]
```

```
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." ++ []))
```

Quiz time!



<https://tinyurl.com/PF2023-C04-Quiz3>

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*

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:

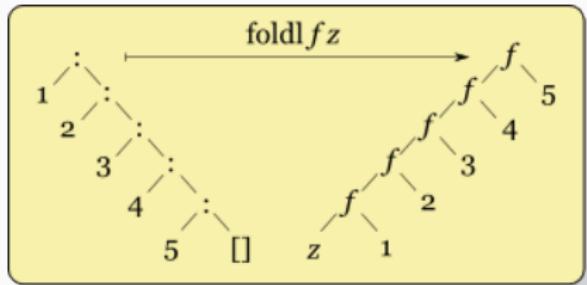
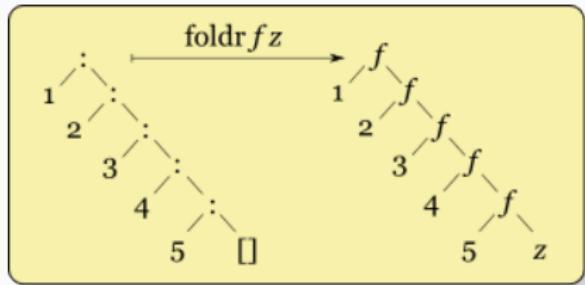
```
Prelude> inf = map (+10) [1..] -- inf nu este evaluat
Prelude> take 3 inf
[11,12,13]
```

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.

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!



<https://tinyurl.com/PF2023-C05-Q1>

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C05

Claudia Chiriță

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
```

```
sucesor :: Season -> Season
```

```
sucesor Spring = Summer
```

```
sucesor Summer = Autumn
```

```
sucesor Autumn = Winter
```

```
sucesor 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 "puncte" 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 și cel de date pot să coincidă

```
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!



<https://tinyurl.com/PF2023-C06-Q1>

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 algebric folosind şabloane

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

Adunarea pe tipul de date algebric:

```
(++) :: 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]
```

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C06

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Înregistrări

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

```
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 are o instanță a clasei **Show**.

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 || elem x ys
```

- definiția folosind funcții de nivel înalt

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

Functia 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'),(3,'d')]
```

True

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

True

Care este tipul functiei my_elem?

Functia my_elem este **polimorfică**.

Definitia functiei este parametrica in tipul de date.

Functia 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 '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 definitii implicite
```

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**!

Sistemul tipurilor in Haskell este complex!

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
```

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
    ...
    ...
```

Puteti verifica folosind comanda `:info` sau `:i` ce contine o anumita clasa de tipuri.

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

O clasă de tipuri este determinată de o mulțime de funcții.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
  -- minimum definition: (==)
  x /= y = not (x == y)
```

Tipurile care aparțin clasei sunt instanțe ale clasei.

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) = (x == x1)
```

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 '<'

Instantiere explicită - exemplu

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

```
Instance Eq Season where
```

```
Spring == Spring = True
```

```
Summer == Summer = True
```

```
Autumn == Autumn = True
```

```
Winter == Winter = True
```

```
_ == _ = False
```

```
Instance Show Season where
```

```
show Spring = "Primavara"
```

```
show Summer = "Vara"
```

```
show Autumn = "Toamna"
```

```
show Winter = "Iarna"
```

Exemplu: liste

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

Exemplu de operație:

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

Comparați cu versiunea folosind notația predefinită:

```
(++) :: [a] -> [a] -> [a]  
[] ++ ys      = ys  
(x:xs) ++ ys = x : (xs ++ ys)
```

Constructori simboluri

```
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
```

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

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

Pe săptămâna viitoare!

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C07

Claudia Chiriță

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Examen parțial - săptămâna viitoare

- **Test grilă pe foaie** în cadrul cursului
- **3 puncte** din nota finală, **15 întrebări grilă, 40 min**
- Nu este obligatoriu și nu se poate reface
- O să afișăm pe Drive repartitia pe ore
- Materiale ajutătoare: suporturile de curs și de laborator, notițe
- **Materiale ajutătoare doar în format fizic!**
- Fără ceasuri, telefoane etc

Functori

Map

În cursurile trecute, am văzut funcția

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

Problema. Putem generaliza această funcție la alte tipuri parametrizate?

Clasa de tipuri Functor

Definiție

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

Dată fiind o funcție $g :: a \rightarrow b$ și ca $:: f a$, $fmap$ produce $cb :: f b$ obținută prin transformarea rezultatelor produse de computația ca folosind funcția g (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 constructorul de tipuri Maybe

```
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 constructorul de tipuri Either e
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 a) (Arboare a)
```

Instantă pentru constructorul de tipuri Arboare

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 în 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)
```

Instanță a clasei Functor care invalidează condiția de conservare a identității:

```
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)
```

Instanță a clasei Functor care validează condiția de conservare a identității:

```
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)
```

Instantă a clasei Functor care invalidează condiția 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)
```

Instantă a clasei Functor care validează condiția 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!



<https://tinyurl.com/PF2023-C08-Quiz1>

Functori applicativi

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?
De exemplu, 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?
De exemplu, 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 $:: m (b \rightarrow c)$

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"]
```

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

Instante – Liste

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

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

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]
```

Quiz time!



<https://tinyurl.com/PF2023-C08-Quiz2>

Pe săptămâna viitoare!