

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C01

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Organizare

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

- Seria 23:
 - Moodle
- Seria 24:
 - Moodle
 - Pagina externa
<https://cs.unibuc.ro/~ddiaconescu/2021/pf/>
- Seria 25:
 - Moodle
- Suporturile de curs si laborator, forumuri, resurse electronice
- Știri legate de curs vor fi postate pe Moodle și pe pagina externă pentru seria 24

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

Notare

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

Condiție de promovabilitate

- Nota finală cel puțin 5 > 4.99

- valorează 3 puncte din nota finală
- nu este obligatoriu
- în săptămâna 7
- se va da pe moodle, online
- prima ora din curs
- va conține întrebări grilă asemănătoare cu cele din curs
- materiale ajutătoare: suportul de curs

- valorează 6 puncte din nota finală
- în sesiune
- acoperă toată materia
- fizic sau online, în funcție de cum se vor desfășura examenele
- durata 2 ore
- va conține întrebări grilă asemănătoare cu cele din curs și exerciții asemănătoare cu cele de la laborator
- materiale ajutătoare: suportul de curs

Activitate laborator

- La sugestia profesorului coordonator al laboratorului, se poate nota activitatea în plus față de cerințele obișnuite.
- Maxim 1 punct (bonus la nota finală)

Extra activitate

- Extra proiecte în timpul semestrului (mai multe detalii spre finalul cursului)
- Maxim 2 puncte (bonus la nota finală)

Activitate laborator

- La sugestia profesorului coordonator al laboratorului, se poate nota activitatea în plus față de cerințele obișnuite.
- Maxim 1 punct (bonus la nota finală)

Extra activitate

- Extra proiecte în timpul semestrului (mai multe detalii spre finalul cursului)
- Maxim 2 puncte (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

- Funcții, recursie, funcții de ordin înalt, tipuri
- Operații pe liste: filtrare, transformare, agregare
- Polimorfism, clase de tipuri, modularizare
- Tipuri de date algebrice - evaluarea expresiilor
- Operațiuni Intrare/ieșire
- Functori, monade

- 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/>

- <https://haskell.foundation/>
- Fundatie independenta si non-profit ce are ca scop imbunatatirea experientei cu limbajul Haskell
- Oferă suport pentru librării, tool-uri, educatie, cercetare
- O sa anuntam mai multe proiecte prin care ii putem ajuta

Nu trișați, cereți-ne ajutorul!



De ce programare funcțională?

Programare declarativă vs. imperativă – Ce vs. cum

Programare imperativă (Cum)

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

Programare declarativă vs. imperativă – Ce vs. cum

Programare imperativă (Cum)

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

Programare declarativă (Ce)

Îi 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** voiam să se întâmple ca rezultat al execuției mașinii.

- limbaje procedurale
- limbaje de programare orientate pe obiecte

Programare declarativă (Ce)

Îi 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** voiam să se întâmple ca rezultat al execuției mașinii.

- limbaje procedurale
- limbaje de programare orientate pe obiecte

Programare declarativă (Ce)

Îi 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

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.

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

- 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"**.

Programare funcțională în limbajul vostru preferat de programare:

- Java 8, C++11, Python, JavaScript, ...
- Funcții anonime (λ -abstracții)
- 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>



Engineers.SG
Meetups Videos

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.



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 ]
```



- Funcțiile sunt valori.
- În loc să modificăm datele existente, calculăm valori noi din valorile existente, folosind funcții

Haskell este un limbaj funcțional pur



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

- Haskell e folosit în proiecte de Facebook, Google, Microsoft etc.
 - Programarea funcțională e din ce în ce mai importantă în industrie
 - mai multe la <https://www.haskell.org/>
- Oferă suport pentru paralelism și concurență.

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 e leneș (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 leneșă poate fi exploatată pentru a reduce timpul de calcul fără a denatura codul

```
firstK k = take k primes
```

- **Haskell e leneș (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 leneșă poate fi exploatată 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
```


Exemplu

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

Elemente de bază. Primii pași

Comentarii

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

Comentarii

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

Identificatori

- șiruri formate din litere, cifre, caracterele `_` și `'` (apostrof)
- identificatorii pentru variabile încep cu literă mică sau `_`
- identificatorii 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
```

Blocuri și indentare.

Blocurile sunt delimitate prin indentare.

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

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
```

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
```

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
```

```
  | ^
```

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

- `=` **nu** este operator de atribuire
- `x = 1` reprezintă o **legătură** (*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

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 expresii **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://www.questionpro.com/t/AT4qgZpBKk>

Seria 24 <https://www.questionpro.com/t/AT4NiZpAK5>

Seria 25 <https://www.questionpro.com/t/AT4qgZpBK5>

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>

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] )]
```

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

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

Tipuri noi definite de utilizator:

```
data RGB = Rosu | Verde | Albastru
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
```

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)
```

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
```

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"]
```

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

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

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 element x numit **capul listei** (*head*) și o listă xs numită **coada listei** (*tail*).

Tipul listă

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

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 în GHCi dacă introducem comanda?

```
Prelude> :t 1
```

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim în GHCi dacă introducem comanda?

```
Prelude> :t 1
```

Răspunsul primit este:

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

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim în GHCi dacă introducem comanda?

```
Prelude> :t 1
```

Răspunsul primit este:

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

Semnificația este următoarea:

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

Tipuri. Clase de tipuri. Variabile de tip

Ce răspuns primim în GHCi dacă introducem comanda?

```
Prelude> :t 1
```

Răspunsul primit este:

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

Semnificația este următoarea:

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

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

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

Prototipul funcției

- numele funcției
- semnătura funcției

`double :: Integer -> Integer`

Funcții în Haskell. Terminologie

Prototipul funcției

- numele funcției
- semnătura funcției

`double :: Integer -> Integer`

Definiția funcției

- numele funcției
- parametrul formal
- corpul funcției

`double elem = elem + elem`

Funcții în Haskell. Terminologie

Prototipul funcției

- numele funcției
- semnătura funcției

`double :: Integer -> Integer`

Definiția funcției

- numele funcției
- parametrul formal
- corpul funcției

`double elem = elem + elem`

Aplicarea funcției

- numele funcției
- parametrul actual (argumentul)

`double 5`

Exemplu: funcție cu două argumente

Prototipul funcției

- numele funcției
- semnătura funcției

add :: Integer -> Integer -> Integer

Definiția funcției

- numele funcției
- parametrii formali
- corpul funcției

add elem1 elem2 = elem1 + elem2

Aplicarea funcției

- numele funcției
- argumentele

add 3 7

Exemplu: funcție cu un argument de tip tuplu

Prototipul funcției

dist :: (Integer, Integer) -> Integer

- numele funcției
- semnatura 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)
```

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)
```

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)
```

Șabloane (patterns)

$x:y = [1,2,3] \text{ -- } x=1 \text{ si } y=[2,3]$

Observați că : este constructorul pentru liste.

Șabloane (patterns)

$x:y = [1,2,3] \text{ -- } x=1 \text{ si } y=[2,3]$

Observați că `:` este constructorul pentru liste.

$(u,v)=('a' ,[(1, 'a') ,(2, 'b')]) \text{ -- } u='a' ,$
 $\text{ -- } v=[(1, 'a') ,(2, 'b')]$

Observați că `(,)` este constructorul pentru tupluri.

Șabloane (patterns)

Definiții folosind șabloane

```
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]`

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 semnatura 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]`

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 semnatura 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]

Quiz time!

Seria 23 <https://www.questionpro.com/t/AT4qgZpBLE>

Seria 24 <https://www.questionpro.com/t/AT4NiZpALo>

Seria 25 <https://www.questionpro.com/t/AT4qgZpBLM>

Pe săptămâna viitoare!