Programare funcțională

Introducere în programarea funcțională folosind Haskell
C02

Ana Iova
Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Funcții

Funcții în Haskell. Terminologie

Prototipul funcției

- **numele funcției**
- signatura funcției

double :: Integer -> Integer

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

Funcții în Haskell. Terminologie

Prototipul funcției **double :: Integer -> Integer**

- **numele funcției**
- signatura funcției

Definiția funcției **double elem = elem + elem**

- **numele funcției**
- parametrul formal
- corpul funcției

Aplicarea funcției **double 5**

- **numele funcției**
- parametrul actual (argumentul)

Exemplu: funcție cu două argumente

Prototipul funcției **add :: Integer -> Integer -> Integer**

- **numele funcției**
- signatura funcției

Definiția funcției **add elem1 elem2 = elem1 + elem2**

- **numele funcției**
- parametrii formali
- corpul funcției

Aplicarea funcției **add 3 7**

- **numele funcției**
- argumentele

Exemplu: funcție cu un argument de tip tuplu

Prototipul funcției **dist :: (Integer, Integer) -> Integer**

- **numele funcției**
- signatura funcției

Definiția funcției **dist (elem1, elem2) = abs (elem1 - elem2)**

- **numele funcției**
- parametrul formal
- corpul funcției

Aplicarea funcției **dist (5, 7)**

- **numele funcției**
- argumentul

Tipuri de funcții

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

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

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

Definirea funcțiilor

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

- Definiție folosind **if**

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

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)
```

Definirea funcțiilor folosind gărzi

Funcția *semn* o putem defini astfel

$$\mathit{semn}\ n = \begin{cases} -1, & \textbf{dacă n} < \textbf{0} \\ 0, & \textbf{dacă n} = \textbf{0} \\ 1, & \textbf{altfel} \end{cases}$$

În Haskell, condițiile devin gărzi:

```
semn  n
      | n < 0          =  -1
      | n = 0          =   0
      | otherwise      =  1
```

Definirea funcțiilor folosind șabloane și ecuații

```
semn :: Integer -> Integer
semn 0 = 0
semn x
  | x > 0      = 1
  | otherwise = -1
```

```
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 funcțiilor folosind șabloane și ecuații

în Haskell, ordinea ecuațiilor este importantă.

Să presupunem că schimbăm ordinii ecuațiilor din definiția factorialului:

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

Ce se întâmplă?

Definirea funcțiilor folosind șabloane și ecuații

în Haskell, ordinea ecuațiilor este importantă.

Să presupunem că schimbăm ordinii 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 șabloane 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 șabloanele sunt _, **True** și **False**.

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

Șablonul _ 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]

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]
```

Quiz time!

Seria 23: https://www.questionpro.com/t/AT4qgZpJPB

Seria 24: https://www.questionpro.com/t/AT4NiZpJFE

Seria 25: https://www.questionpro.com/t/AT4qgZpJPM

Liste

Liste (slide din C01)

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

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

Liste (slide din C01)

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

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

Definiție recursivă. O **listă** este

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

Definirea listelor. Operații

Intervale și progresii

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

Definirea listelor. Operații

Intervale și progresii

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

Operații

```
Prelude> [1,2,3] !! 2
3
Prelude> "abcd" !! 0
'a'
Prelude> [1,2] ++ [3]
[1,2,3]
Prelude> import Data.List
```

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

[E(x)| x <- [x1,...,xn], P(x)]

```
Prelude> let xs = [0..10]
Prelude> [x | x <- xs, even x]
```

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

[E(x)| x <- [x1,...,xn], P(x)]

```
Prelude> let xs = [0..10]
Prelude> [x | x <- xs, even x]
[0,2,4,6,8,10]
```

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

[E(x)| x <- [x1,...,xn], P(x)]

```
Prelude> let xs = [0..10]
Prelude> [x | x <- xs, even x]
[0,2,4,6,8,10]

Prelude> let xs = [0..6]
Prelude> [(x,y) | x <- xs, y <- xs,  x + y == 10]
```

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

[E(x)| x <- [x1,...,xn], P(x)]

```
Prelude> let xs = [0..10]
Prelude> [x | x <- xs, even x]
[0,2,4,6,8,10]

Prelude> let 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)| x <- [x1,. . . ,xn], P(x)]

Folosirea lui `let` pentru declarații locale:

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

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

[E(x)| x <- [x1,. . . ,xn], P(x)]

Folosirea lui `let` pentru declarații locale:

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

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

[E(x)| x <- [x1,...,xn], P(x)]

Folosirea lui `let` pentru declarații locale:

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

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

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

[E(x)| x <- [x1,...,xn], P(x)]

Folosirea lui `let` pentru declarații locale:

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

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

zip xs ys

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

zip xs ys

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

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

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

zip xs ys

Observați diferența!

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

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

Lenevire (Lazyness)

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

```
Prelude> head []
*** Exception: Prelude.head: empty list
```

Lenevire (Lazyness)

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

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> let x = head []
Prelude> let f a = 5
Prelude> f x
5
```

Lenevire (Lazyness)

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

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> let x = head []
Prelude> let f a = 5
Prelude> f x
5
Prelude> [1,head [],3] !! 0
1
Prelude> [head [],3] !! 1
*** Exception: Prelude.head: empty list
```

Liste infinite

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

```
Prelude> let natural = [0..]
Prelude> take 5 natural
[0,1,2,3,4]
```

Liste infinite

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

```
Prelude> let natural = [0..]
Prelude> take 5 natural
[0,1,2,3,4]

Prelude> let evenNat = [0,2..] -- progresie infinita
Prelude> take 7 evenNat
[0,2,4,6,8,10,12]
```

Liste infinite

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

```
Prelude> let natural = [0..]
Prelude> take 5 natural
[0,1,2,3,4]

Prelude> let evenNat = [0,2..] -- progresie infinita
Prelude> take 7 evenNat
[0,2,4,6,8,10,12]

Prelude> let ones = [1,1..]
Prelude> let zeros = [0,0..]
Prelude> let both = zip ones zeros
Prelude> take 5 both
[(1,0),(1,0),(1,0),(1,0),(1,0)]
```

Șabloane (patterns)

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

Observați că : este constructorul pentru liste.

Șabloane (patterns)

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

Observați că : este constructorul pentru liste.

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

Observați că (,) este constructorul pentru tupluri.

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
```

Șabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

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

Observați:

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

Ce s-a întâmplat?

- x:y este un șablon pentru liste
- potrivirea dintre x:y și [1,2,3] a avut ca efect:
  - "deconstrucția" valorii [1,2,3] în 1:[2,3]
  - legarea lui x la 1 și a lui y la [2,3]

Șabloane (patterns) pentru liste

Definiții folosind șabloane

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

- x:xs se potrivește cu liste nevide

Șabloane (patterns) pentru liste

Definiții folosind șabloane

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

- x:xs se potrivește cu liste nevide

Atenție!

Șabloanele sunt definite folosind constructori. De exemplu, operația de concatenare pe liste este `(++) :: [a] -> [a] -> [a]`, dar

`[x] ++ [1] = [2,1]` nu va avea ca efect legarea lui x la 2;

încercând să evaluăm x vom obține un mesaj de eroare:

```
Prelude> [x] ++ [1] = [2,1]
Prelude> x
error: ...
```

Șabloanele sunt liniare

În Haskell șabloanele sunt liniare, adică o variabilă apare cel mult odată.

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

```
x:x:[1] = [2,2,1]
ttail (x:x:t) = t
foo x x = x^2
```

error: Conflicting definitions for x

Șabloanele sunt liniare

În Haskell șabloanele sunt liniare, adică o variabilă apare cel mult odată.

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

```
x:x:[1] = [2,2,1]
ttail (x:x:t) = t
foo x x = x^2
```

error: Conflicting definitions for x

O soluție este folosirea gărzilor:

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

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

Quiz time!

Seria 23: https://www.questionpro.com/t/AT4qgZpJPU

Seria 24: https://www.questionpro.com/t/AT4NiZpJFf

Seria 25: https://www.questionpro.com/t/AT4qgZpJPe

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

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
```

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
```

Funcții ca operatori

```
Prelude> mod 5 2
1
Prelude> 5 `mod` 2
1
```

Funcții ca operatori

```
Prelude> mod 5 2
1
Prelude> 5 `mod` 2
1
```

- 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
  ```

Funcții ca operatori

```
Prelude> mod 5 2
1
Prelude> 5 `mod` 2
1
```

- 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
  ```

```
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
```

Precedență și asociativitate

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

Precedență și asociativitate

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

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

Asociativitate

Operatorul - asociativ la stanga

```
5 - 2 - 1 == (5 - 2) - 1          -- /= 5 - (2 - 1)
```

Asociativitate

Operatorul - asociativ la stanga

```
5 - 2 - 1 == (5 - 2) - 1          -- /= 5 - (2 - 1)
```

Operatorul : asociativ la dreapta

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

Asociativitate

Operatorul - asociativ la stanga

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

Operatorul : asociativ la dreapta

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

Operatorul ++ asociativ la dreapta

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

$l1 ++ l2 ++ l3 ++ l4 ++ l5 == l1 ++ (l2 ++ (l3 ++ (l4 ++ l5)))$

Secțiuni ("operator sections")

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

Matematic, ele corespund aplicării parțiale a funcției op.

Secțiuni ("operator sections")

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

Matematic, ele corespund aplicării parțiale a funcției op.

Aplicarea parțială.

Fie $f : A \times B \to C$ o funcție. În mod uzual scriem

$f(a, b) = c$ unde $a \in A$, $b \in B$ și $c \in C$.

Pentru $a \in A$ și $b \in B$ (arbitrare, fixate) definim

$f_a : B \to C$, $f_a(b) = c$ dacă și numai dacă $f(a, b) = c$,

$f^b : A \to C$, $f^b(a) = c$ dacă și numai dacă $f(a, b) = c$.

Funcțiile f_a și f_b se obțin prin aplicarea parțială a funcției f.

Secțiuni ("operator sections")

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

- secțiunile lui ++ sunt (++ e) și (e ++)

Secțiuni ("operator sections")

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

- secțiunile lui ++ sunt (++ e) și (e ++)

```
Prelude> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
Prelude> (++ " world!") "Hello" -- atentie la paranteze
"Hello world!"
```

Secțiuni ("operator sections")

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

- secțiunile lui ++ sunt (++ e) și (e ++)

```
Prelude> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
Prelude> (++ " world!") "Hello" -- atentie la paranteze
"Hello world!"
Prelude> ++ " world!" "Hello"
error
```

Secțiuni ("operator sections")

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

- secțiunile lui ++ sunt (++ e) și (e ++)

```
Prelude> :t (++ " world!")
(++ " world!") :: [Char] -> [Char]
Prelude> (++ " world!") "Hello" -- atentie la paranteze
"Hello world!"
Prelude> ++ " world!" "Hello"
error
```

- secțiunile lui <-> sunt (<-> e) și (e <->), unde

```
Prelude> let x <-> y = x-y+1  -- definit de utilizator
Prelude> :t (<-> 3)
(<-> 3) :: Num a => a -> a
Prelude> (<-> 3) 4
2
```

Secțiuni

- secțiunile operatorului (:)

Secțiuni

- secțiunile operatorului (:)

```
Prelude> (2:)[1,2]
[2,1,2]
Prelude> (:[1,2]) 3
[3,1,2]
```

Secțiuni

- secțiunile operatorului (:)

```
Prelude> (2:)[1,2]
[2,1,2]
Prelude> (:[1,2]) 3
[3,1,2]
```

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
```

Compunerea funcțiilor — operatorul .

Matematic. Date fiind $f : A \to B$ și $g : B \to C$, compunerea lor, notată $g \circ f : A \to C$, 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 precedența 0.

```
($) :: (a -> b) -> a -> 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
```

Funcții de nivel înalt

Funcții anonime

Funcții anonime = lambda expresii

\x1 x2 ··· xn -> expresie

Funcții anonime

Funcții anonime = lambda expresii

\x1 x2 ··· xn -> expresie

```
Prelude> (\x -> x + 1) 3
4
Prelude> inc = \x -> x + 1
Prelude> add = \x y -> x + y
Prelude> aplic = \f x -> f x

Prelude> map (\x -> x+1) [1,2,3,4]
[2,3,4,5]
```

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

Funcțiile pot fi folosite ca argumente pentru alte funcții.

Funcțiile sunt valori

Exemplu:

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

Funcțiile sunt valori

Exemplu:

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

- definiția cu lambda expresii

  ```
  flip f = \x y -> f y x
  ```

- definiția folosind șabloane

  ```
  flip f x y = f y x
  ```

- **flip** ca valoare de tip funcție

  ```
  flip = \f x y -> f y x
  ```

Funcții de ordin înalt

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

Funcții de ordin înalt

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

Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Funcții de ordin înalt

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

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Un exemplu mai complicat:

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

Funcții de ordin înalt

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

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Un exemplu mai complicat:

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

Funcții de ordin înalt

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

```
Prelude> map (* 3) [1,3,4]
[3,9,12]
```

Un exemplu mai complicat:

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

În acest caz:

- primul argument este o sectiune a operatorului ($)
- al doilea argument este o lista de functii

  map ($ x) [f_1,. . ., f_n] == [f_1 x,. . ., f_n x]

Funcții de ordin înalt

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

Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Funcții de ordin înalt

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

Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Compunere și aplicare

```
Prelude> let f l = map (* 3) (filter (>= 2) l)
Prelude> f [1,3,4]
```

Funcții de ordin înalt

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

Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Compunere și aplicare

```
Prelude> let f l = map (* 3) (filter (>= 2) l)
Prelude> f [1,3,4]
[9, 12]                    -- [ x * 3 | x <- [1,3,4], x >=2 ]
```

Funcții de ordin înalt

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

Prelude> filter (>= 2) [1,3,4]
[3,4]
```

Compunere și aplicare

```
Prelude> let f l = map (* 3) (filter (>= 2) l)
Prelude> f [1,3,4]
[9, 12]                    -- [ x * 3 | x <- [1,3,4], x >=2 ]
```

Definiția compozițională *(pointfree style)*:

```
f = map (* 3) . filter (>=2)
```

Quiz time!

Seria 23: https://www.questionpro.com/t/AT4qgZpJkQ

Seria 24: https://www.questionpro.com/t/AT4NiZpJGy

Seria 25: https://www.questionpro.com/t/AT4qgZpJkT

Pe săptămâna viitoare!