

# Programare declarativă

## Introducere în programarea funcțională folosind Haskell

Ioana Leuștean  
Ana Cristina Turlea

Departamentul de Informatică, FMI, UB  
ioana@fmi.unibuc.ro  
ana.turlea@fmi.unibuc.ro

- 1 Terminologie. Forma curry
- 2 Operatori. Secțiuni
- 3 Definirea funcțiilor. Șabloane
- 4 Funcții de nivel înalt

## Terminologie. Forma curry

# Funcții în Haskell. Terminologie

## Prototipul funcției

- **numele funcției**
- **signatura 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: adunarea a doi întregi

## 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** (**2**, **5**)

- **numele funcției**
- **argumentul**

# Funcții în matematică

- Fie  $f : A \times B \rightarrow C$  o funcție. În mod uzual scriem  $f(x, y) = z$  unde  $x \in A$ ,  $y \in B$  și  $z \in C$ .

- Pentru  $x \in A$  (arbitrar, fixat) definim

$$f_x : B \rightarrow C, f_x(y) = z \text{ dacă și numai dacă } f(x, y) = z.$$

Funcția  $f_x$  se obține prin **aplicarea parțială** a funcției  $f$ .

În mod similar definim *aplicarea parțială* pentru orice  $y \in B$

$$f^y : A \rightarrow C, f^y(x) = z \text{ dacă și numai dacă } f(x, y) = z.$$

# Funcții în matematică

## Exemplu

$A = \text{Int}, B = C = \text{String}$

$$f(x, y) = \begin{cases} z, & |y| \geq x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x \leq 0 \end{cases}$$

- Fie  $x \in \text{Int}$  arbitrar, fixat. Atunci  $f_x : \text{String} \rightarrow \text{String}$  și
  - dacă  $x \leq 0$ , atunci  $f_x(y) = ""$  oricare  $y$
  - dacă  $x > 0$  atunci  $f_x(y) = \begin{cases} z, & |y| \geq x, |z| = x, y = zw \\ y, & 0 < |y| < x \end{cases}$
- Fie  $y \in \text{String}$  arbitrar, fixat. Atunci  $f^y : \text{Int} \rightarrow \text{String}$  și

$$f^y(x) = \begin{cases} z, & |y| \geq x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x \leq 0 \end{cases}$$



# Funcții în matematică

- Fie  $f : A \times B \rightarrow C$  o funcție. În mod uzual scriem  $f(x, y) = z$  unde  $x \in A$ ,  $y \in B$  și  $z \in C$ .
- Pentru  $x \in A$  (arbitrar, fixat) definim  $f_x : B \rightarrow C$ ,  $f_x(y) = z$  dacă și numai dacă  $f(x, y) = z$ .
- Dacă notăm  $B \rightarrow C \stackrel{\text{not}}{=} \{h : B \rightarrow C \mid h \text{ funcție}\}$  observăm că  $f_x \in B \rightarrow C$  pentru orice  $x \in A$ .
- Asociem lui  $f$  funcția

$$cf : A \rightarrow (B \rightarrow C), \quad cf(x) = f_x$$

Observăm că pentru fiecare element  $x \in A$ , funcția  $cf$  întoarce ca rezultat funcția  $f_x \in B \rightarrow C$ , adică

$$cf(x)(y) = z \text{ dacă și numai dacă } f(x, y) = z$$

## Forma **curry**

Vom spune că funcția  $cf$  este *forma curry* a funcției  $f$ .

## De la matematică la Haskell

Funcția  $f : \text{Int} \times \text{String} \rightarrow \text{String}$

$$f(x, y) = \begin{cases} z, & |y| \geq x, |z| = x, y = zw \\ y, & 0 < |y| < x \\ "", & x \leq 0 \end{cases}$$

poate fi definită în Haskell astfel:

```
f :: (Int, String) -> String
f (n,s) = take n s
```

Observăm că:

```
Prelude> let cf = curry f
Prelude> :t cf
cf :: Int -> String -> String
Prelude> f(1,"abc")
"a"
Prelude> cf 1 "abc"
"a"
```

# 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 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  $(\dots ((f\ x_1)\ x_2) \dots x_n)$ .

# Funcții și mulțimi

## Teoremă

Mulțimile  $(A \times B) \rightarrow C$  și  $A \rightarrow (B \rightarrow C)$  sunt echipotente.

## Observație

Funcțiile **curry** și **uncurry** din Haskell stabilesc bijecția din teoremă:

```
Prelude> :t curry
```

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

```
Prelude> :t uncurry
```

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

## 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]**

## 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`

# 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 ' '

```
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**  
**True**

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

# Asociativitate

Operatorul - asociativ la stanga

$$5 - 2 - 1 == (5 - 2) - 1 \quad \text{---} \quad /= 5 - (2 - 1)$$

Operatorul : asociativ la dreapta

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

Operatorul ++ asociativ la dreapta

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

$$[] ++ ys = ys$$

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

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

Care este complexitatea aplicării operatorului ++?

- liniară în lungimea primului argument

## Secțiuni ("operator sections")

Secțiunile operatorului binar **op** sunt **(op e)** și **(e op)**.  
 Matematic, ele corespund aplicării parțiale a funcției **op**.

- secțiunile lui **||** sunt **(|| e)** și **(e ||)**

```
Prelude> :t (|| True)
```

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

```
Prelude> (|| True) False -- atentie la paranteze
```

```
True
```

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

```
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
```

```
8
```

# 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
```

# Funcții anonime și secțiuni

Funcții anonime = lambda expresii

**$\backslash x_1 x_2 \dots x_n \rightarrow \text{expresie}$**

**Prelude**>  $(\backslash x \rightarrow x + 1) 3$

4

**Prelude**>  $\text{inc} = \backslash x \rightarrow x + 1$

**Prelude**>  $\text{add} = \backslash x y \rightarrow x + y$

**Prelude**>  $\text{aplic} = \backslash f x \rightarrow f x$

Secțiunile sunt definite prin lambda expresii:

$(x +) = \backslash y \rightarrow x + y$

$(+ y) = \backslash x \rightarrow x + y$

# Compunerea funcțiilor — operatorul .

## Matematic

Date fiind  $f : A \rightarrow B$  și  $g : B \rightarrow C$ , compunerea 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> z=1
```

```
Prelude> t=2
```

```
Prelude> sqrt (z^2+t^2)
```

```
2.23606797749979
```

```
Prelude> x = 1 :: Integer
```

```
Prelude> y = 3 :: Integer
```

```
Prelude> sqrt fromIntegral (x^2+y^2)
```

```
<interactive>:33:1: error:
```

```
Prelude> sqrt (fromIntegral (x^2+y^2))
```

```
3.1622776601683795
```

```
Prelude> sqrt . fromIntegral (x^2+y^2)
```

```
<interactive>:36:1: error:@*
```

```
Prelude> (sqrt . fromIntegral) (x^2+y^2)
```

```
3.1622776601683795
```

# Operatorul \$

Operatorul (\$) are precedența 0.

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

$$f \$ x = f x$$

```
Prelude> sqrt 3 + 4 +9
```

```
14.732050807568877
```

```
Prelude> sqrt (3 + 4 +9)
```

```
4.0
```

```
Prelude> sqrt $ 3 + 4 +9
```

```
4.0
```

Operatorul (\$) este asociativ la dreapta.

```
Prelude> sqrt $ fromIntegral $ x^2+y^2
```

```
3.1622776601683795
```



## Definirea funcțiilor. Șabloane

# Definirea funcțiilor folosind **if**

- analiza cazurilor folosind expresia "if"

```
semn : Integer -> Integer
semn n = if n < 0 then (-1)
          else if n=0 then 0
                else 1
```

- definiție recursivă în care analiza cazurilor folosește expresia "if"

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

# Definirea funcțiilor folosind **gărzi**

Funcția *semn* o putem defini astfel

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

În Haskell, condițiile devin **gărzi**:

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

# Definirea funcțiilor folosind **gărzi**

Funcția *fact* o putem defini astfel

$$fact\ n = \begin{cases} 1, & \text{dacă } n = 0 \\ n * fact(n - 1), & \text{altfel} \end{cases}$$

În Haskell, condițiile devin **gărzi**:

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

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 pentru valori pozitive.

# 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.

## Șabloane (patterns) pentru liste

Listele sunt construite folosind constructorii (:) și []

$[1, 2, 3] == 1:[2, 3] \quad \text{--} == \quad 1:2:[3] == 1:2:3:[]$

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

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: ...**

## Șabloane pentru tupluri

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

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

- Definiii 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
```

## 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

```
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 = ...
```

## Funcții de nivel înalt

# Funcțiile sunt valori

Funcțiile — „cetățeni de rangul I”

Funcțiile sunt valori,  
care pot fi trimise ca argument sau întoarse ca rezultat

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

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

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

# Funcții de ordin înalt

## filter

```
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)
```

# Funcții de ordin înalt

## foldr și foldl

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr o z [a1, a2, a3, ..., an] ==
    a1 'o' (a2 'o' (a3 'o' (... (an 'o' z) ...)))
```

```
Prelude> foldr (*) 1 [1,3,4]
12                -- product [1,3,4]
```

```
foldl :: (b -> a -> b) -> b -> t a -> b
foldl o z [a1, a2, a3, ..., an] ==
    ( ... (((z 'o' a1) 'o' a2) 'o' a3) 'o' ... an)
```

```
Prelude> foldl (flip (:)) [] [1,3,4]
[4,3,1]  -- de ce? intelegeti modul de functionare!
```



# Filtrare, transformare, agregare

## Suma pătratelor elementelor pozitive

- Folosind descrieri de liste și funcții de agregare standard

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

- Folosind funcții auxiliare

```
f xs = foldr (+) 0 (map sqr (filter pos xs))
  where
    sqr x = x * x
    pos x = x > 0
```

- Folosind funcții anonime

```
f :: [Int] -> Int
f xs = foldr (+) 0
  (map (\x -> x * x) (filter (\x -> x > 0) xs))
```

# Filtrare, transformare, agregare

## Suma pătratelor elementelor pozitive

- Folosind secțiuni și operatorul \$ (parametru explicit)

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

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

- Definiție compozițională (pointfree style)

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

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

Pe săptămâna viitoare!