

# 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
- 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

`add :: Integer -> Integer -> Integer`

- numele funcției
- semnătura 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
- semnătura 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

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

În Haskell, condițiile devin gărzi:

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

## Definirea funcțiilor folosind 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 semnelor = 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 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/AT4qgZpJPB>

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

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

## Liste

---

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

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

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*).

## Definirea listelor. Operații

### Intervale și progresii

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



# Definirea listelor. Operații

## Intervale și progresii

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

## Operații

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

```
Prelude> "abcd" !! 0
'a'
```

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

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

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

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

```
Prelude> let xs = [0..10]
```

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

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

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

```
Prelude> let xs = [0..10]
```

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

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

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

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

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

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

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

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

Folosirea lui **let** pentru declarații locale:

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

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

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

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

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

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

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

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"
```

```
Prelude> let xs = ['A'..'Z']
```

```
Prelude> [x | (i,x) <- [1..] `zip` xs, even i]
```

```
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')]
```

## 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 (Lazyiness)

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> [1, head [], 3] !! 1  
*** Exception: Prelude.head: empty list
```

## Liste infinite

Drept consecință a **evaluării leneșe**, 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 leneșe**, 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 leneșe**, 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)

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

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
```

## 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  $(++) :: [a] \rightarrow [a] \rightarrow [a]$ , 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>

**Pe săptămâna viitoare!**