

# Programare declarativă<sup>1</sup>

## Operatori, Funcții (din nou), Recursie (din nou)

Ioana Leuștean  
Traian Florin Șerbănuță

Departamentul de Informatică, FMI, UNIBUC  
traian.serbanuta@unibuc.ro

20 octombrie 2017

---

<sup>1</sup>bazat pe cursul Informatics 1: Functional Programming de la University of Edinburgh

# Operatori

# Operatorii sunt funcții

## Operatori Booleeni

**not** ' :: **Bool** → **Bool**

# Operatorii sunt funcții

## Operatori Booleeni

**not' :: Bool -> Bool**

**not' True = False**

**not' False = True**

# Operatorii sunt funcții

## Operatori Booleeni

**not' :: Bool -> Bool**

**not' True = False**

**not' False = True**

**(&&&) :: Bool -> Bool -> Bool**

# Operatorii sunt funcții

## Operatori Booleeni

**not' :: Bool -> Bool**

**not' True = False**

**not' False = True**

**(&&&) :: Bool -> Bool -> Bool**

**True &&& b = b**

**False &&& \_ = False**

# Funcțiile sunt operatori

## Operatori aritmetici

```
Prelude> mod 5 2
```

```
1
```

```
Prelude> 5 'mod' 2
```

```
1
```

# Funcțiile sunt operatori

## Operatori aritmetici

```
Prelude> mod 5 2
```

```
1
```

```
Prelude> 5 'mod' 2
```

```
1
```

```
divide :: Int -> Int -> Bool
```



# Funcțiile sunt operatori

## Operatori aritmetici

```
Prelude> mod 5 2
```

```
1
```

```
Prelude> 5 'mod' 2
```

```
1
```

```
divide :: Int -> Int -> Bool
```

```
x 'divide' y = y 'mod' x == 0
```

# Funcțiile sunt operatori

## Operatori aritmetici

```
Prelude> mod 5 2
```

```
1
```

```
Prelude> 5 'mod' 2
```

```
1
```

```
divide :: Int -> Int -> Bool
```

```
x 'divide' y = y 'mod' x == 0
```

```
apartine :: Int -> [Int] -> Bool
```

# Funcțiile sunt operatori

## Operatori aritmetici

```
Prelude> mod 5 2
```

```
1
```

```
Prelude> 5 'mod' 2
```

```
1
```

```
divide :: Int -> Int -> Bool
```

```
x 'divide' y = y 'mod' x == 0
```

```
apartine :: Int -> [Int] -> Bool
```

```
x 'apartine' [] = False
```

```
x 'apartine' (y:xs) = x == y || (x 'apartine' xs)
```

# Operatorii sunt funcții

## Operatorii în Haskell

- sunt definiți folosind numai „simboluri” (ex: `*!*`)
- au două argumente
- sunt apelați folosind notația infix

# Operatorii sunt funcții

## Operatorii în Haskell

- sunt definiți folosind numai „simboluri” (ex: `*!*`)
- au două argumente
- sunt apelați folosind notația infix

- Operatori predefiniți

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

`True || _ = True`

`False || b = b`

# Operatorii sunt funcții

## Operatorii în Haskell

- sunt definiți folosind numai „simboluri” (ex: `*!*`)
- au două argumente
- sunt apelați folosind notația infix

- Operatori predefiniți

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

`True || _ = True`

`False || b = b`

- Operatori definiți de utilizator

`(&&&) :: Bool -> Bool -> Bool` *-- atentie la paranteze*

`True &&& b = b`

`False &&& _ = False`

# Precedență și asociativitate

**Prelude**>  $3 + 5 * 4 : [6] ++ 8 - 2 + 3 : [2] == [23, 6, 9, 2] || \text{True} == \text{False}$

# 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'

# Declararea precedenței și a modului de grupare

**infix, infixl, infixr**

```
(<+>) :: Int -> Int -> Int
```

```
x <+> y = x + y + 1
```

```
*Main> 1 <+> 2 * 3 <+> 4
```

# Declararea precedenței și a modului de grupare

**infix, infixl, infixr**

```
(<+>) :: Int -> Int -> Int
```

```
x <+> y = x + y + 1
```

```
*Main> 1 <+> 2 * 3 <+> 4
```

```
32
```

```
--(1 <+> 2) * (3 <+> 4)
```

Precedența implicită este 9 (maximă)

# Declararea precedenței și a modului de grupare

**infix, infixl, infixr**

**infixl 6 <+>**

**(<+>) :: Int -> Int -> Int**

**x <+> y = x + y + 1**

**\*Main> 1 <+> 2 \* 3 <+> 4**

**13**

# Declararea precedenței și a modului de grupare

**infix, infixl, infixr**

```
infixl 6 <+>
```

```
(<+>) :: Int -> Int -> Int
```

```
x <+> y = x + y + 1
```

```
*Main> 1 <+> 2 * 3 <+> 4
```

```
13
```

```
egal :: Float -> Float -> Bool
```

```
x 'egal' y = abs(x - y) <= 0.001
```

```
*Main> 1 / 32 'egal' 1 / 33
```

# Declararea precedenței și a modului de grupare

**infix, infixl, infixr**

**infixl 6 <+>**

**(<+>) :: Int -> Int -> Int**

**x <+> y = x + y + 1**

**\*Main> 1 <+> 2 \* 3 <+> 4**

**13**

**egal :: Float -> Float -> Bool**

**x 'egal' y = abs(x - y) <= 0.001**

**\*Main> 1 / 32 'egal' 1 / 33**

**Eroare de sintaxă**

**--(1 / (32 'egal' 1)) / 33**

# Declararea precedenței și a modului de grupare

**infix, infixl, infixr**

```
infixl 6 <+>
```

```
(<+>) :: Int -> Int -> Int
```

```
x <+> y = x + y + 1
```

```
*Main> 1 <+> 2 * 3 <+> 4
```

```
13
```

```
infix 4 'egal'
```

```
egal :: Float -> Float -> Bool
```

```
x 'egal' y = abs(x - y) <= 0.001
```

```
*Main> 1 / 32 'egal' 1 / 33
```

```
True
```

# Precedență și asociativitate

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



# De ce?

De ce este operatorul - asociativ la stanga?

## De ce?

De ce este operatorul - asociativ la stanga?

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

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

## De ce?

De ce este operatorul `-` asociativ la stanga?

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

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

De ce este operatorul `:` asociativ la dreapta?

# De ce?

De ce este operatorul `-` asociativ la stanga?

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

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

De ce este operatorul `:` asociativ la dreapta?

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

## De ce?

De ce este operatorul `-` asociativ la stanga?

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

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

De ce este operatorul `:` asociativ la dreapta?

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

De ce este operatorul `++` asociativ la dreapta?

## De ce?

De ce este operatorul `-` asociativ la stanga?

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

De ce este operatorul `:` asociativ la dreapta?

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

Care este complexitatea aplicării operatorului `++`?

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

$$[] ++ ys = ys$$

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

De ce este operatorul `++` asociativ la dreapta?

## De ce?

De ce este operatorul `-` asociativ la stanga?

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

De ce este operatorul `:` asociativ la dreapta?

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

Care este complexitatea aplicării operatorului `++`?

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

$[] ++ ys = ys$

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

liniară în lungimea primului argument

De ce este operatorul `++` asociativ la dreapta?

## De ce?

De ce este operatorul `-` asociativ la stanga?

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

De ce este operatorul `:` asociativ la dreapta?

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

Care este complexitatea aplicării operatorului `++`?

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

$[] ++ ys = ys$

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

liniară în lungimea primului argument

De ce este operatorul `++` asociativ la dreapta?

Vrem ca lungimea primului argument să fie cât mai mică

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



## Funcții(din nou)

## Definirea funcțiilor folosind șabloane ("patterns")

```
wfact 0 =1  
wfact (succ n) = (succ n) * (wfact n)
```

```
Prelude> :t succ  
succ :: Enum a => a -> a
```

# Definirea funcțiilor folosind șabloane ("patterns")

Ce este greșit?

```
wfact 0 =1
wfact (succ n) = (succ n) * (wfact n)
```

```
Prelude> :t succ
succ :: Enum a => a -> a
```

**succ** nu este constructor!

Forma corectă

```
fact 0 =1
fact n = n * fact (n -1)
```

# Definirea funcțiilor folosind șabloane ("patterns")

```
wlen [] = 0  
wlen [x] = 1  
wlen (xs ++ ys) = (wlen xs) + (wlen ys)
```

# Definirea funcțiilor folosind șabloane ("patterns")

Ce este greșit?

```
wlen [] = 0
wlen [x] = 1
wlen (xs ++ ys) = (wlen xs) + (wlen ys)
```

**++ nu este constructor!**

Forma corectă

```
length [] = 0
length (x:xs) = 1 + (length xs)
```

# Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

```
take :: Int -> [a] -> [a]
```

```
Prelude> take 3 [1,2,3,4,5,6]
[1,2,3]
```

<b>take</b>	0	$\_$	=	<code>[]</code>
<b>take</b>	$\_$	<code>[]</code>	=	<code>[]</code>
<b>take</b>	n	<code>(x:xs)</code>	=	<code>x : take (n-1) xs</code>

# Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

```
take :: Int -> [a] -> [a]
```

```
Prelude> take 3 [1,2,3,4,5,6]
[1,2,3]
```

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

- șabloanele se definesc folosind constructori
- se face potrivirea între parametrii actuali ai funcției și șabloane
- ordinea de scriere a ecuațiilor este importantă

# Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

Ordinea de scriere a ecuațiilor este importantă!

<b>take</b>	0	$\_$	=	<code>[]</code>
<b>take</b>	$\_$	<code>[]</code>	=	<code>[]</code>
<b>take</b>	n	<code>(x:xs)</code>	=	<code>x : take (n-1) xs</code>
<b>take1</b>	$\_$	<code>[]</code>	=	<code>[]</code>
<b>take1</b>	0	$\_$	=	<code>[]</code>
<b>take1</b>	n	<code>(x:xs)</code>	=	<code>x : take1 (n-1) xs</code>



# Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

Ordinea de scriere a ecuațiilor este importantă!

```
take 0 _ = []
take _ [] = []
take n (x:xs) = x : take (n-1) xs
```

```
take1 _ [] = []
take1 0 _ = []
take1 n (x:xs) = x : take1 (n-1) xs
```

```
*Main> take 0 undefined
```

```
[]
```

```
*Main> take1 0 undefined
```

```
Exception: Prelude.undefined
```

```
*Main> take1 undefined []
```

```
[]
```

```
*Main> take undefined []
```

```
Exception: Prelude.undefined
```

# Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

Ordinea de scriere a ecuațiilor este importantă!

```

take  0      _      =  []
take  _      []      =  []
take  n      (x:xs)  =  x : take (n-1) xs
  
```

```

*Main> take 0 undefined
[]
*Main> take undefined []
Exception: Prelude.undefined
  
```

Care este explicația?

# Definirea funcțiilor folosind șabloane ("patterns")

<https://www.haskell.org/tutorial/patterns.html>

Ordinea de scriere a ecuațiilor este importantă!

```
take 0      _      = []
take _      []      = []
take n      (x:xs)  = x : take (n-1) xs
```

```
*Main> take 0 undefined
[]
*Main> take undefined []
Exception: Prelude.undefined
```

Care este explicația?

- potrivirea dintre `_` și **undefined** nu forțează evaluarea
- potrivirea dintre `0` și **undefined** forțează evaluarea

# Definirea funcțiilor

## error și undefined

```
Prelude> :t error
error :: [Char] -> a
Prelude> :t undefined
undefined :: a
```

# Definirea funcțiilor

## error și undefined

```
Prelude> :t error
error :: [Char] -> a
Prelude> :t undefined
undefined :: a
```

```
fact 0 =1
fact n = if (n >= 1)
         then n * fact (n -1)
         else undefined           -- error "nu se calculeaza"
```

# Definirea funcțiilor

tratarea cazurilor de eroare

<http://book.realworldhaskell.org/read/functional-programming.html>

## Definirea funcției **head**

- folosind **length**

```
myHead xs = if length xs > 0
             then head xs
             else undefined
```

- folosind **null**

```
myHead xs = if not (null xs)
             then head xs
             else undefined
```

# Definirea funcțiilor

tratarea cazurilor de eroare

<http://book.realworldhaskell.org/read/functional-programming.html>

## Definirea funcției **head**

- folosind **length**

```
myHead xs = if length xs > 0
             then head xs
             else undefined
```

- folosind **null**

```
myHead xs = if not (null xs)
             then head xs
             else undefined
```

Care variantă este mai bună?

# Definirea funcțiilor

tratarea cazurilor de eroare

<http://book.realworldhaskell.org/read/functional-programming.html>

## Definirea funcției **head**

- folosind **length**

```
myHead xs = if length xs > 0
             then head xs
             else undefined
```

- folosind **null**

```
myHead xs = if not (null xs)
             then head xs
             else undefined
```

Care variantă este mai bună?

Varianta cu **null**, pentru a calcula **length** trebuie parcursă toată lista!



# Definirea funcțiilor

## Gărzi

```
fact 0 =1
fact n
  | (n >= 1) = n * fact (n -1)
  | otherwise = undefined    -- otherwise == True
```

Ordinea gărzilor are importanță.

```
tanar n
  | (n >= 60) = "nu asa de tanar"
  | (n >= 40) = "tanar"
  | (n >= 18) = "foarte tanar"
  | (n >= 14) = "adolescent"
  | (n > 0)   = "copil"
  | otherwise = undefined
```

## Recursie (din nou)

# Generarea [m..n]

```
Prelude> [3..7]
```

```
[3,4,5,6,7]
```

```
Prelude> enumFromTo 3 7
```

```
[3,4,5,6,7]
```

[m..n] este o notație pentru **enumFromTo** m n

```
enumFromTo :: Integer -> Integer -> [Integer]
```

# Generarea [m..n]

**Prelude**> [3..7]

[3,4,5,6,7]

**Prelude**> **enumFromTo** 3 7

[3,4,5,6,7]

[m..n] este o notație pentru **enumFromTo** m n

**enumFromTo** :: **Integer** -> **Integer** -> [**Integer**]

**enumFromTo** m n | m > n = []  
                   | **otherwise** = m : **enumFromTo** (m + 1) n

# Generarea [m..]

[m..] este o notație pentru **enumFrom m**

**enumFrom :: Integer -> [Integer]**

## Generarea [m..]

[m..] este o notație pentru **enumFrom m**

```
enumFrom :: Integer -> [Integer]  
enumFrom m = m : enumFrom (m + 1)
```

### Exemplu de rulare

```
enumFrom 4  
= 4 : enumFrom 5  
= 4 : 5 : enumFrom 6  
= 4 : 5 : 6 : enumFrom 7  
= 4 : 5 : 6 : 7 : enumFrom 8  
= .....
```

# Zip

Zip împerechează (în ordine, câte două) elementele a două liste

**zip** :: [a] -> [b] -> [(a,b)]

# Zip

Zip împerechează (în ordine, câte două) elementele a două liste

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

```
zip [] ys = []
```

```
zip xs [] = []
```

```
zip (x:xs) (y:ys) = (x, y) : zip xs ys
```

## Exemplu de rulare

```
zip [0,1,2] "abc"
= (0,'a') : zip [1,2] "bc"
= (0,'a') : ((1,'b') : zip [2] "c")
= (0,'a') : ((1,'b') : ((2,'c') : zip [] ""))
= (0,'a') : ((1,'b') : ((2,'c') : []))
= [(0,'a'),(1,'b'),(2,'c')]
```



## Zip cu liste infinite

```

zip :: [a] -> [b] -> [(a,b)]
zip [] ys = []
zip xs [] = []
zip (x:xs) (y:ys) = (x, y) : zip xs ys

```

### Exemplu de rulare (leneșă)

```

zip [0..] "abc"
= zip (0:[1..]) "abc"
= zip (0:[1..]) ('a':"bc")
= (0,'a') : zip [1..] "bc"
= (0,'a') : ((1,'b') : zip [2..] "c")
= (0,'a') : ((1,'b') : ((2,'c') : zip [3..] ""))
= (0,'a') : ((1,'b') : ((2,'c') : zip (3:[4..]) ""))
= (0,'a') : ((1,'b') : ((2,'c') : []))
= [(0,'a'),(1,'b'),(2,'c')]

```

# Produs scalar

Pentru doi vectori  $\bar{a}$  și  $\bar{b}$  de aceeași lungime, produsul scalar este  $\sum_i a_i * b_i$

dot :: Num a => [a] -> [a] -> a

# Produs scalar

Pentru doi vectori  $\bar{a}$  și  $\bar{b}$  de aceeași lungime, produsul scalar este  $\sum_i a_i * b_i$

```
dot :: Num a => [a] -> [a] -> a
dot xs ys = sum [x * y | (x,y) <- xs 'zip' ys]
```

## Exemplu de rulare

```
[1,2,3] 'dot' [4,5,6]
= sum [x * y | (x,y) <- [1,2,3] 'zip' [4,5,6]]
= sum [x * y | (x,y) <- [(1,4),(2,5),(3,6)]]
= sum [1*4,2*5,3*6]
= sum [4,10,18]
= 720
```

# Search

search caută toate pozițiile dintr-o listă pe care apare un element dat.

search :: **Eq** a => [a] -> a -> [**Int**]

# Search

search caută toate pozițiile dintr-o listă pe care apare un element dat.

```
search :: Eq a => [a] -> a -> [Int]
```

```
search xs x = [i | (i,y) <- [0..] 'zip' xs, y == x]
```

## Exemplu de rulare

```
search "abac" 'a'
```

```
= [i | (i,y) <- [0..] 'zip' "abac", y == 'a']
```

```
= [i | (i,y) <- [(0,'a'),(1,'b'),(2,'a'),(3,'c')], y == 'a']
```

```
= [0 | 'a' == 'a'] ++ [1 | 'b' == 'a'] ++ [2 | 'a' == 'a'] ++  
   [3 | 'c' == 'a']
```

```
= [0,2]
```