

Programare funcțională

Introducere în programarea funcțională folosind Haskell
C03

Ana Iova

Denisa Diaconescu

Departamentul de Informatică, FMI, UB

Anunt - examen parțial

- valorează 3 puncte din nota finală
- durata ~~45 min~~ 40 min
- în săptămâna 7, in cadrul cursului
- test grila pe foaie
- nu este obligatoriu si nu se poate reface
- ~~cursul 7 se va tine online pe Teams~~
- va conține întrebări grilă asemănătoare cu cele din curs
- materiale ajutătoare: suporturile de curs si de laborator

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

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`

Prelude> mod 5 2

1

Prelude> 5 `mod` 2

1

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`, `rem`, `quot`		
6	+, -		
5			:, ++
4		==, /=, <, <=, >, >=, `elem`, `notElem`	
3			&&
2			
1	>>, >>=		
0			`, \$!, `seq`

Operatorul - asociativ la stânga

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

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**).

Secțiunile lui **(++) sunt **(++ e)** și **(e ++)****

```
Prelude> :t (++)
```

```
(++) :: [a] -> [a] -> [a]
```

```
Prelude> :t (++ " world!")
```

```
(++ " world!") :: [Char] -> [Char]
```

```
Prelude> (++ " world!") "Hello "
```

```
"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> x <-> y = x-y+1 *-- definit de noi*

Prelude> :t (<-> 3)

(<-> 3) :: Num a => a -> a

Prelude> (<-> 3) 4

2

Secțiuni

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 de nivel înalt

Funcții anonime

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

Funcțiile pot fi folosite ca argumente pentru alte funcții.

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

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

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> :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.

$$(\$) :: (a \rightarrow b) \rightarrow a \rightarrow b$$
$$f \$ x = f x$$

```
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
```


Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZu3zB>

Seria 24: <https://questionpro.com/t/AT4NiZu3KF>

Seria 25: <https://questionpro.com/t/AT4qgZu3zE>

Procesarea fluxurilor de date: Map, Filter, Fold

Transformarea fiecărui element dintr-o listă - map

Exemplu - Pătrate

Definiți o funcție care pentru o listă de numere întregi dată ridică la pătrat fiecare element din listă.

Soluție descriptivă

```
squares :: [Int] -> [Int]
squares xs = [ x * x | x <- xs ]
```

Soluție recursivă

```
squares :: [Int] -> [Int]
squares [] = []
squares (x:xs) = x*x : squares xs
```

```
Prelude> squares [1,-2,3]
[1,4,9]
```

Exemplu - Coduri ASCII

Transformați un șir de caractere în lista codurilor ASCII ale caracterelor.

Soluție descriptivă

```
ords :: [Char] -> [Int]
ords xs = [ ord x | x <- xs ]
```

Soluție recursivă

```
ords :: [Char] -> [Int]
ords []      = []
ords (x:xs) = ord x : ords xs
```

```
Prelude> ords "a2c3"
[97,50,99,51]
```

Funcția map

Date fiind o funcție de transformare și o listă,
aplicați funcția fiecărui element al unei liste date.

Soluție descriptivă

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

Soluție recursivă

```
map :: (a -> b) -> [a] -> [b]  
map f []      = []  
map f (x:xs) = f x : map f xs
```

Exemplu — Pătrate

Soluție descriptivă

```
squares :: [Int] -> [Int]
squares xs = [ x * x | x <- xs ]
```

Soluție recursivă

```
squares :: [Int] -> [Int]
squares [] = []
squares (x:xs) = x*x : squares xs
```

Soluție folosind map

```
squares :: [Int] -> [Int]
squares xs = map sqr xs
              where sqr x = x * x
```

Exemplu — Coduri ASCII

Soluție descriptivă

```
ords :: [Char] -> [Int]
ords xs = [ ord x | x <- xs ]
```

Soluție recursivă

```
ords :: [Char] -> [Int]
ords []      = []
ords (x:xs) = ord x : ords xs
```

Soluție folosind map

```
ords :: [Char] -> [Int]
ords xs = map ord xs
```


Funcții de ordin înalt

```
map :: (a -> b) -> [a] -> [b]
```

```
map f l = [f x | x <- l]
```

```
Prelude> map ($) [(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

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

Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvALb>

Seria 24: <https://questionpro.com/t/AT4NiZvAxq>

Seria 25: <https://questionpro.com/t/AT4qgZvALf>

Selectarea elementelor dintr-o listă - filter

Exemplu - Selectarea elementelor pozitive dintr-o listă

Definiți o funcție care selectează elementele pozitive dintr-o listă.

Soluție descriptivă

```
positives :: [Int] -> [Int]
positives xs = [ x | x <- xs, x > 0 ]
```

Soluție recursivă

```
positives :: [Int] -> [Int]
positives [] = []
positives (x:xs) | x > 0 = x : positives xs
                  | otherwise = positives xs
```

```
Prelude> positives [1,-2,3]
[1,3]
```

Exemplu - Selectarea cifrelor dintr-un șir de caractere

Definiți o funcție care selectează cifrele dintr-un șir de caractere.

Soluție descriptivă

```
digits :: [Char] -> [Char]
digits xs = [ x | x <- xs, isDigit x ]
```

Soluție recursivă

```
digits :: [Char] -> [Char]
digits [] = []
digits (x:xs) | isDigit x = x : digits xs
              | otherwise = digits xs
```

```
Prelude> digits "a2c3"
"23"
```

Date fiind un predicat (funcție booleană) și o listă, selectați elementele din listă care satisfac predicatul.

Soluție descriptivă

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

Soluție recursivă

```
filter :: (a -> Bool) -> [a] -> [a]  
filter p [] = []  
filter p (x:xs) | p x = x : filter p xs  
                | otherwise = filter p xs
```

Exemplu — Selectarea elementelor pozitive dintr-o listă

Soluție descriptivă

```
positives :: [Int] -> [Int]
positives xs = [ x | x <- xs, x > 0 ]
```

Soluție recursivă

```
positives :: [Int] -> [Int]
positives [] = []
positives (x:xs) | x > 0 = x : positives xs
                  | otherwise = positives xs
```

Soluție folosind filter

```
positives :: [Int] -> [Int]
positives xs = filter pos xs
              where pos x = x > 0
```

Exemplu — Selectarea cifrelor dintr-un șir de caractere

Soluție descriptivă

```
digits :: [Char] -> [Char]
digits xs = [ x | x <- xs, isDigit x ]
```

Soluție recursivă

```
digits :: [Char] -> [Char]
digits [] = []
digits (x:xs) | isDigit x = x : digits xs
               | otherwise = digits xs
```

Soluție folosind filter

```
digits :: [Char] -> [Char]
digits xs = filter isDigit xs
```


Quiz time!

Seria 23: <https://questionpro.com/t/AT4qgZvALr>

Seria 24: <https://questionpro.com/t/AT4NiZvAxv>

Seria 25: <https://questionpro.com/t/AT4qgZvALs>

Pe săptămâna viitoare!